MASTER EN SCIENCES INFORMATIQUES

Conception d'un logiciel automatisant le contrôle et l'analyse de modèles TVL

Faber, Paul

Award date:
2010

Awarding institution:
Universite de Namur

Conception d'un logiciel automatisant le contrôle et l'analyse de modèles TVL

Paul Faber

2009 - 2010

Mémoire présenté en vue de l'obtention du grade de Maître en informatique

Abstract. Nowadays, software product lines are an efficient way to create a family of software sharing common features and also differing in their own specific particularities. The crucial task in a SPL is the management and the modelling of the variability. Variability with a high abstraction level can be modelled through a diagram of features. In this case, a feature can represent, for example, a functionality or a characteristic of one or several software of the SPL. Nevertheless, those diagrams show gaps which prevent them to respond to the needs of the industrial world. In order to make up these gaps, The FUNDP's LIEL team developed TVL, a textual language based on this concept of features. The purpose is to propose it to companies to meet their needs. However, at this stage, TVL couldn't be fully exploited because there aren't any tools to automate the analysis of the TVL models and to control them. That is the reason why a software called "TVL parser" is presented in this thesis. This software has three major functionalities. Firstly, it can do a syntactic and semantic analysis of a TVL model. Secundly, it can generate the normal form of this model (just using a subset of the TVL generators). Thirdly, it can calculate different operations about this model using a SAT solver under specific conditions. It can ,among others, determine if a model is satisfiable or not.

Keywords : software product lines, variability, feature diagram, TVL, model transformation, model checking

Résumé. Actuellement, les software product lines ou lignes de produits logiciels représentent un moyen efficace de produire une famille de logiciels. Dans cette dernière, les logiciels partagent des caractéristiques communes mais possèdent aussi chacun leurs propres particularités. La gestion et la modélisation de la variabilité qui existe entre ces logiciels est une tâche cruciale au sein d'une SPL. Les diagrammes de features permettent de modéliser cette variabilité à un haut niveau d'abstraction. Ici, une feature peut représenter une fonctionnalité, une caractéristique, ... d'un ou plusieurs des logiciels de la SPL. Cependant, ces diagrammes possèdent des lacunes les empêchant d'être en adéquation avec les besoins du monde industriel. Dans ce contexte, l'équipe LIEL des FUNDP a développé TVL, un langage textuel aussi basé sur le concept de features. Le but de ce langage est de combler les lacunes des diagrammes de features et de proposer aux entreprises une solution adaptée à leurs exigences. Toutefois, présenté ainsi, TVL ne peut pas être pleinement exploité. En effet, il n'existe aucun outil automatisant l'analyse et le contrôle de modèles TVL. Dès lors, afin de combler ce vide, dans ce travail, un logiciel nommé "parseur TVL" est présenté. Ce dernier possède trois fonctionnalités majeures. Premièrement, il est capable d'analyser syntaxiquement et sémantiquement un modèle TVL. Deuxièmement, il peut générer la forme normalisée de ce modèle (n'employant qu'un sous-ensemble des constructeurs de TVL). Troisièmement, en employant un solveur SAT et à certaines conditions, il est capable de calculer diverses opérations concernant le modèle. Il peut notamment déterminer si oui ou non le modèle est satisfiable.

Remerciements

Je voudrais tout d'abord remercier Patrick Heymans et Andreas Classens, les copromoteurs de ce mémoire. Je leur suis reconnaissant d'avoir accepté de superviser ce travail et et de toujours avoir su me guider et m'épauler efficacement.

Je tiens aussi à remercier les membres de l'équipe LIEL pour l'accueil qu'ils m'ont offert durant mon stage. Je tiens tout particulièrement à exprimer ma gratitude envers Quentin Boucher qui m'a aussi fourni une aide précieuse lors de la rédaction de ce mémoire.

Je remercie aussi ma famille et tous les amis qui m'ont aidé durant la rédaction de ce travail. Je remercie spécialement ma mère qui a toujours su me soutenir, même dans les moments les plus difficiles.

Enfin, même s'il n'est plus parmi nous, je tiens aussi à remercier mon père qui m'a laissé cette énergie qui lui était propre, me permettant de toujours continuer à avancer et à progresser.

Table des matières

Table des figures

Chapitre 1

Introduction

Durant ces trente dernières années, les développeurs de logiciels ont assisté à une évolution des exigences de leurs clients. Dans les années quatre-vingts, un logiciel était souvent destiné à un usage unique. Les clients devaient fréquemment "se contenter" des logiciels existants. Même si l'un d'entre eux désirait un programme uniquement pour un sous-ensemble de ses fonctionnalités, il était forcé d'acheter le logiciel complet et donc de payer un prix conséquent. Aujourd'hui, les besoins des clients ont évolué. Un client ne se contente plus de ce qu'on lui propose, il veut un logiciel adapté, personnalisé selon ses besoins. Dès lors, afin de rester compétitives, les entreprises ont dû s'adapter et proposer des logiciels les plus en adéquation possible avec les besoins d'un client ou d'un ensemble de clients.

Cependant, développer des logiciels personnalisés de manière traditionnelle n'est pas rentable. Pour chaque logiciel, il faut créer un nouveau projet : recruter une équipe, analyser les exigences, rédiger un cahier des charges, ... Vu que le logiciel s'adresse à un nombre restreint de clients, les frais de développement sont difficilement amortissables. Afin de résoudre ce problème de non-rentabilité, les entreprises ont adopté une nouvelle stratégie de production, inspirée des lignes de production classiques : les "software product lines" (SPLs) ou lignes de produits logiciels.

Une SPL représente un ensemble de logiciels possédant des points communs (fonctionnalités, composants, ...). Ces logiciels sont construits à l'aide d'artéfacts (librairies, use case, ...) et chaque artéfact peut être réutilisé, moyennant certains paramétrages, dans plusieurs logiciels de la SPL. C'est notamment grâce à cette réutilisation que les SPLs permettent de produire des logiciels personnalisés de manière rentable. Cependant, la rentabilité n'est pas le seul avantage des SPLs, ces dernières permettent aussi de créer des logiciels de qualité, de réduire les délais de commercialisation et de faciliter la gestion de la complexité.

Dans une SPL, un artéfact doit être conçu de manière "flexible", afin de pouvoir être réutilisé dans plusieurs logiciels. Chaque artéfact doit donc pouvoir varier, s'adapter en fonction du logiciel dans lequel il est employé. Au sein d'une SPL, il existe donc une notion de "variabilité". Cette dernière est un des principes-clés d'une SPL. Si elle n'est pas gérée correctement, elle peut provoquer l'échec de la ligne de produits. Dès lors, durant toute la durée de vie de la SPL, il est primordial de la gérer efficacement. Heureusement, des méthodes et des techniques ont été développées afin d'assister les ingénieurs dans cette

tâche. A un haut niveau d'abstraction, il existe notamment les diagrammes de features.

Dans un tel diagramme, de manière simplifiée, une feature représente une caractéristique (fonctionnalité, performance, ...) d'un ou plusieurs des logiciels de la SPL. Ces features sont structurées à l'aide d'un arbre ou d'un graphe. Un diagramme reprend donc l'ensemble des caractéristiques des logiciels de la SPL. Grâce à certains mécanismes (relations entre les features, contraintes, cardinalités, ...), la variabilité de cette SPL y est représentée. En sélectionnant une certaine combinaison de features, un acteur (client, développeur, ...) construit une configuration correspondant à un des logiciels de la SPL. Toutefois, cette sélection est "réglementée", elle doit respecter un ensemble de contraintes, notamment celles exprimées à travers la structure du diagramme.

Malheureusement, les diagrammes de features possèdent certains défauts qui font qu'ils ne peuvent pas toujours être utilisés pour modéliser des SPLs d'entreprises. Graphiquement, représenter les centaines de features que peut compter une SPL industrielle est quasi impossible. De plus, sans un outil spécifique, explorer ou manipuler ce genre de diagrammes peut constituer un réel défi. Par conséquent, en utilisant des diagrammes de features, l'acteur devient dépendant de la technologie. Enfin, la sémantique de ce type de diagramme est rarement définie, ce qui rend leur exploitation compliquée voire parfois hasardeuse. Par exemple, sans la définition de la sémantique, il n'est pas possible de contrôler automatiquement (à l'aide d'un logiciel) le contenu d'un diagramme.

Afin de combler les lacunes des diagrammes de features et de fournir une méthode applicable dans le monde industriel, TVL, un langage textuel de modélisation basé sur les features, a été développé par l'équipe LIEL des FUNDP [1]. Grâce à sa syntaxe proche du C, TVL peut être utilisé par n'importe quelle personne ayant un minimum de connaissances en informatique. De plus, à l'aide de ses différents mécanismes de modularisation, les acteurs peuvent représenter un modèle, comprenant de nombreuses features, de manière concise et personnalisée. Finalement, l'aspect textuel de TVL garantit qu'il ne nécessite pas d'outils dédiés. N'importe quel éditeur de texte, même le plus basique, peut être utilisé pour créer et modifier un modèle TVL.

Toutefois, présenté ainsi, TVL ne peut pas être pleinement exploité par les entreprises. Pour un développeur, créer ou modifier un modèle TVL ne pose aucun problème. Cependant, au moment de contrôler ce modèle, les choses se compliquent. En effet, pour un être humain, vérifier manuellement (sans l'aide d'un logiciel) un modèle composé de centaines de features, de centaines de contraintes, ... peut se révéler être une tâche complexe fort gourmande en temps. De manière similaire, calculer manuellement des opérations par rapport à ce genre de modèle peut être impossible. Dès lors, afin d'automatiser le contrôle et l'analyse de modèles TVL, le support d'un logiciel est nécessaire. Sans l'aide d'un programme, les avantages de TVL risqueraient d'être occultés par les inconvénients liés au contrôle des modèles et les industriels pourraient alors ne trouver aucun intérêt à utiliser ce langage. Or, **actuellement, il n'existe aucun outil automatisant le contrôle et l'analyse de modèles TVL**.

Dès lors, dans ce mémoire, **dans le but de solutionner ce problème, un logiciel est proposé afin d'automatiser l'analyse de modèles et de donner un aperçu pratique des avantages du langage TVL**. Ce logiciel

1. Site web officiel des FUNDP : `http://www.fundp.ac.be/`

permettra notamment de contrôler un modèle TVL et de calculer si oui ou non ce dernier est satisfiable, c'est-à-dire, déterminer s'il existe au moins une combinaison de features respectant l'ensemble des contraintes du modèle.

Structure du mémoire

Ce mémoire est composé de trois parties comme expliqué ci-dessous.

Partie I : Cette première partie permettra de contextualiser la problématique. Le Chapitre 2 sera donc consacré aux SPLs et illustrera en quoi il est intéressant de mener des recherches les concernant. Ensuite, le Chapitre 3 permettra d'affiner le contexte en se focalisant sur un des concepts-clés des SPLs : la variabilité. Il présentera les différents aspects de cette dernière et la façon de la modéliser à l'aide des diagrammes de features. Enfin, le Chapitre 4 exposera les lacunes majeures des diagrammes de features et le langage introduit par l'équipe LIEL afin de les combler : TVL.

Partie II : Cette deuxième partie sera consacrée à la problématique et à la solution qui y est apportée. Le Chapitre 4 introduira la problématique (l'absence d'outils automatisant le contrôle et l'analyse de modèles TVL) et la solution proposée : le parseur TVL. Ensuite, les Chapitres 6 à 8 permettront de présenter en détail les différentes fonctionnalités du logiciel.

Partie III : Cette troisième et dernière partie permettra de clôturer ce mémoire. Le Chapitre 9 exposera les limites de TVL et proposera de l'étendre en permettant de modéliser explicitement les différents aspects de la variabilité présentés dans le Chapitre 3. Ensuite, le Chapitre 10 permettra d'évaluer le parseur TVL et de dresser ses perspectives d'avenir. Enfin, le Chapitre 11 contiendra la conclusion de ce mémoire.

Le domaine des SPLs possédant une terminologie spécifique, afin de simplifier la lecture de ce mémoire, l'auteur de ce dernier a conçu un glossaire. L'ensemble des définitions qu'il contient peut être trouvé à la page 121.

A titre informatif, le rapport du stage lié au sujet de ce mémoire est disponible dans l'annexe C, page 139.

Première partie

Contextualisation

Chapitre 2

Le paradigme des SPLs

Ce premier chapitre va permettre de présenter le contexte global de la problématique : les lignes de produits logiciels ou software product lines (SPLs). Parallèlement, il va aussi illustrer en quoi les SPLs sont avantageuses et donc ce pourquoi il est intéressant de mener des recherches les concernant. Afin de faciliter la compréhension, la Section 2.1 sera consacrée aux lignes de production classiques. Cela permettra d'introduire par la suite les SPLs grâce à une analogie construite sur deux niveaux : premièrement, sur l'évolution des besoins qu'ont connue les marchés de l'automobile et du logiciel. Deuxièmement, sur les concepts similaires qui régissent les lignes de production classiques et les SPLs. La Section 2.2 sera donc dédiée au développement du marché du logiciel tandis que la suivante, la 2.3, traitera des concepts-clés des SPLs. Ensuite, la Section 2.4 permettra de présenter les activités nécessaires au développement et à l'utilisation d'une SPL. Enfin, la dernière section exposera les avantages et les facteurs d'échecs des SPLs.

Ce chapitre a principalement été rédigé à l'aide de deux ouvrages de référence : *Software Product Line Engineering : Foundations, Principles and Techniques* de Pohl, Böckle et van der Linden [22]. Et *Software Product Lines : Practices and Patterns* de Clements et Northrop [10].

2.1 La création des lignes de production et la personnalisation de masse

En 1913, après plusieurs essais et expérimentations, Henry Ford invente la première chaîne de montage automobile [1]. La standardisation des pièces, l'application du travail à la chaîne et la mise en place des chaînes de montage lui permettent de faire passer sa production de Ford modèle T de 80000 unités/an en 1912 à 230000 unités/an en 1914, la cadence de production étant alors d'une voiture/minute. La Ford Motor Company est alors capable de **produire** des automobiles **en masse**. Ce nouveau type de production permet de baisser le prix du modèle T de 850$ en 1909 à 550$ en 1915. Cette chute des prix fait décoller les ventes de 69762 unités en 1911 à 501462 unités en 1915 [33]. La production

1. Il faut tout de même noter que Ford s'est inspiré de processus utilisés dans d'autres industries, notamment dans l'agroalimentaire et dans la fabrication de conserves [20]

de masse permet donc d'augmenter la cadence de production tout en diminuant le prix du produit. Tout cela, sans perte de bénéfice pour l'organisation.

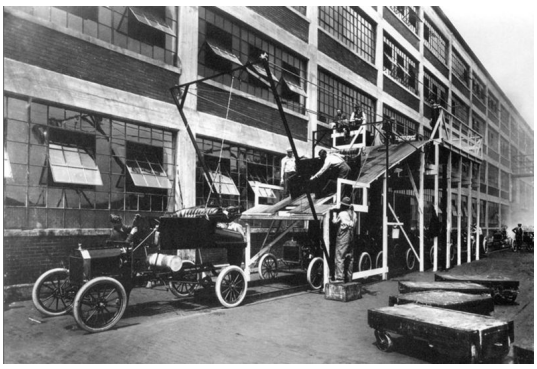

FIGURE 2.1 – Une Ford modèle T sortant d'une chaîne de montage.

Dans les années vingt, suite au développement des réseaux routiers, les exigences des clients se diversifient, tout le monde ne veut plus rouler dans la même auto. Certains veulent se démarquer socialement, ils réclament des voitures plus belles et plus confortables [1] . La production en masse (à laquelle plusieurs fabricants automobiles ont dorénavant recours) ne suffit donc plus. Les industriels se rendent compte qu'il est désormais impératif de prendre en compte les besoins de leurs clients. Ceci va déboucher sur un processus de production plus évolué : la **personnalisation de masse** ("mass customisation"). Cette nouvelle manière de produire peut être vue comme un moyen de fabriquer en masse des produits qui répondent aux besoins individuels d'un groupe de clients, d'un segment de marché.

Au premier abord, la personnalisation de masse n'est pas souvent une sinécure pour une entreprise, elle réclame des investissements technologiques, ce qui augmente le prix du produit et/ou diminue la marge de profit. Pour les sociétés, il était donc devenu vital de trouver une réponse à ce problème. La solution, appliquée par plusieurs d'entre elles, fut l'utilisation de **plate-formes** communes ("common platform").

Une plate-forme représente l'ensemble des biens qui pourront être réutilisés dans les différents produits d'une même gamme. Dans le secteur de l'automobile, ces biens peuvent être des pièces mécaniques, des plans de production, des logiciels embarqués,... Avant de concevoir la plate-forme, il faut donc **prévoir et planifier** quels biens seront utilisés dans quels produits. Ces biens doivent donc être assez **flexibles** que pour s'adapter aux différents produits dans lesquels ils seront utilisés. Les coûts de développement d'une plate-forme sont souvent élevés. Cependant, ces investissement sont rentabilisés grâce à la **réutilisation** des biens (il n'est plus nécessaire de développer le bien, il faut juste quelques fois le **"paramétrer"** en fonction du produit voulu). Pour une société, une plate-forme est donc un moyen rentable de proposer une gamme de produits répondant aux exigences spécifiques de clients. L'ensemble des biens produits à partir d'une même plateforme constitue une **ligne de produits** (**"product line"**, PL) ou une **famille de produits** (**"product family"**, PF).

2.2 Le développement logiciel en 1980 et l'évolution des besoins

Dans les années quatre-vingts, la plupart des logiciels sont développés de manière traditionnelle : un programme peut être développé en ne partant de rien ("from scratch") ou à partir d'un logiciel précédemment conçu. Chaque software nécessite un projet individuel, toute une équipe de spécialistes lui est donc attribuée. Dans la majorité des cas, il est destiné à un usage unique (ou à un seul client). **Un logiciel n'est pas considéré comme quelque chose de variable ou de personnalisable**. Même si un client ne désire le logiciel que pour une de ses fonctionnalités spécifiques, il est obligé de l'acheter entièrement.

Au niveau des logiciels embarqués ("embedded software", logiciel utilisé à l'intérieur d'un périphérique : téléphones portables, ...), la situation est quelque peu différente. Un logiciel est une façon de proposer différentes versions (avec différentes fonctions) d'un même appareil. Il est quelquefois plus facile et plus rentable de remplacer certaines fonctions, qui nécessitaient auparavant du hardware, par du software. Cependant, le processus de personnalisation est loin d'être efficace, le logiciel est souvent personnalisé très tard dans le projet. Ceci a donc pour conséquence de complexifier la tâche des développeurs, qui finissent par produire des logiciels quelquefois peu fiables. A cette époque, **peu de personnes attachent de l'importance au processus de production**.

Tout comme le marché automobile, le marché des logiciels va être confronté à une évolution des besoins des clients, engendrant aussi l'évolution des processus de production.

Aujourd'hui, les exigences en matière de logiciel ont évolué. Pour rester compétitives, les compagnies sont obligées d'y prêter attention. Les logiciels doivent maintenant répondre efficacement aux besoins spécifiques des clients. Pour un même programme, un client pourra, par exemple, vouloir toutes les fonctionnalités tandis qu'un autre n'en désirera que certaines. Produire des logiciels personnalisés à l'aide des méthodes classiques s'est révélé fort coûteux, les organisations ont donc dû créer de nouveaux processus de production, plus adaptés et plus rentables. Parallèlement, la complexité des programmes a augmenté de façon colossale. Une application peut maintenant faire plusieurs centaines de mégaoctets et contenir plusieurs millions de lignes de code. Pour les développeurs, la gestion de la complexité est devenue un problème majeur.

Actuellement, les SPLs représentent un moyen efficace de réponse à ces nouveaux problèmes et exigences. Elles permettent de produire rapidement, à faible prix, des applications personnalisées répondant aux exigences des clients, d'appareils ou de logiciels. Grâce à la notion de variabilité, les SPLs permettent aussi de gérer plus facilement la complexité.

2.3 Concepts-clés des SPLs

La majeure partie des notions et du vocabulaire des SPLs ont été repris aux PLs classiques. Les concepts majeurs sont donc les mêmes tout en ayant été adaptés au domaine des logiciels. Les sous-sections suivantes reprennent chacun de ces principes et la Figure 2.2 (située sur la page suivante) les illustre.

2.3.1 La personnalisation de masse

Dans le domaine des SPLs, le concept-clé est aussi la **personnalisation de masse**, c'est-à-dire la production en masse de logiciels qui répondent aux exigences individuelles de clients, d'appareils ou encore d'autres logiciels. Les produits créés peuvent être des logiciels embarqués, des logiciels classiques ou des composants logiciels. Par exemple, une SPL pourrait servir à créer les systèmes d'exploitation d'une gamme de PDA. Chaque version de l'OS ("Operating System", le système d'exploitation) serait personnalisée en fonction du modèle du PDA, elle contiendrait les modules logiciels correspondant aux fonctionnalités de l'appareil. La Figure 2.2 illustre cet exemple.

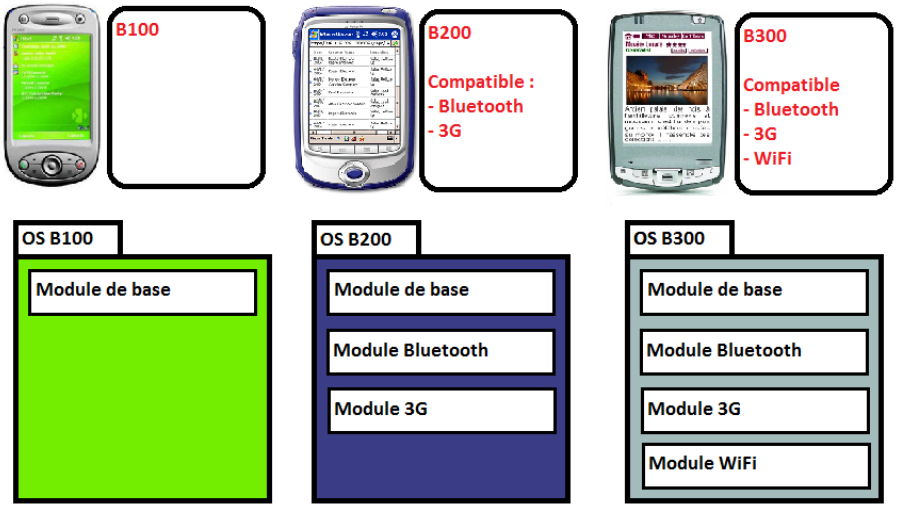

FIGURE 2.2 – Les OS correspondant aux différents modèles de PDA.

Dans cet exemple, le modèle B100 n'est compatible avec aucun standard. Son OS ne contiendra donc que le module de base. Le modèle au-dessus, le B200, est compatible Bluetooth et 3G. L'OS qui y sera installé devra donc contenir le module de base mais aussi les modules correspondant au Bluetooth et à la 3G. Enfin, en plus d'être compatible Bluetooth et 3G, le modèle B300 est aussi capable d'utiliser les réseaux WiFi. Son OS sera composé du module de base, du module 3G, du module Bluetooth ainsi que du module WiFi.

Afin d'appliquer efficacement ce style de production au domaine des logiciels, il est nécessaire d'adopter une structure de développement (plate-forme) tout en respectant une certaine stratégie (réutilisation, planification,...).

2.3.2 Plate-formes et réutilisation

Avec les processus de développement classiques, la personnalisation de masse entraînerait des problèmes et des coûts supplémentaires. Il a donc fallu trouver un moyen pour rendre ce mode de production rentable – tout comme cela a été jadis le cas pour les lignes de produits "classiques" – ce moyen fut incarné par les plate-formes. Dans le domaine des SPLs, une plate-forme est constituée d'un ensemble d'artéfacts. Un artéfact peut être un composant logiciel, une ar-

chitecture, un test,... Chaque logiciel d'une SPL résulte de l'assemblage d'un ensemble d'artéfacts. Si un artéfact est contenu dans la plate-forme, c'est qu'il a été prévu de le réutiliser dans plusieurs applications de la SPL. Sauf indications contraires, toute partie spécialement développée pour un logiciel ne sera pas intégrée à la plate-forme. En effet, vu que cette partie est destinée à un usage unique, il ne sera pas intéressant de la maintenir (adaptation aux avancées technologiques,...) car elle ne sera de toute façon probablement plus utilisée. Dans l'exemple, la plate-forme sera au moins constituée du module de base, du module Bluetooth, du module 3G, des plans de test correspondant à ces modules,... Elle ne contiendra normalement pas le module WiFi, ce dernier ayant été spécialement développé pour le modèle B300. Cependant, il se peut qu'il soit prévu de sortir dans les prochains mois un nouveau modèle de PDA intégrant une antenne WiFi. Dès lors, il pourra être intéressant d'intégrer le module WiFi dans la plate-forme. De cette manière, il pourra être réutilisé pour le nouveau modèle.

La réutilisation des artéfacts est donc un autre principe-clé des SPLs. Elle permet de faire des économies lors de la production de chaque produit. En effet, pour chaque produit, il n'est plus nécessaire de développer les artéfacts, il "suffit" de les paramétrer en fonction des exigences. La principale tâche des développeurs n'est plus la programmation mais l'intégration des différents artéfacts pour construire les produits. Dans l'exemple, le module de base est réutilisé dans chaque logiciel, ce qui permet de réaliser de substantielles économies. A l'inverse, le module WiFi a dû être spécialement développé pour le B300, il sera donc moins rentable car il ne sera pas intégré dans les autres PDA.

2.3.3 Planification et stratégie

Avant de lancer le développement d'une SPL, il faut d'abord définir l'ensemble des produits qui seront développables. Il faut alors s'intéresser à toutes les caractéristiques et exigences qu'ils auront en commun pour après uniquement se soucier des fonctionnalités spécifiques de chacun d'eux. Il ne faut pas voir la SPL comme une multitude de produits mais plutôt comme un tout. La création de la plate-forme n'est donc pas une tâche facile, il faut planifier à l'avance l'ensemble des artéfacts nécessaires aux produits. Le développement d'un artéfact ne se fait pas au hasard et de façon opportuniste. S'il est créé, c'est parce qu'il sera réutilisable dans au moins deux produits. Dans l'exemple, il a été prévu que la SPL serait composée de trois produits logiciels différents pour lesquels il faudrait développer trois artéfacts (base, Bluetooth, 3G). Il a aussi été prévu qu'il faudrait élaborer un module spécifique, le module WiFi.

2.3.4 Flexibilité

Comme précisé plus haut, chaque artéfact de la plate-forme doit être réutilisable dans plusieurs produits. Par conséquent, tous les artéfacts doivent être développés de façon à être facilement intégrables dans les produits pour lesquels ils ont été prévus (voire même pour des produits futurs). Des artéfacts pas assez flexibles ou peu génériques risqueraient de compromettre le succès de la SPL (pour chaque produit, il faudrait "re-développer" l'artéfact, faisant perdre le caractère avantageux de la réutilisation). Dans l'exemple, le module de base sera

par exemple développé de manière à pouvoir s'intégrer dans les trois types de PDA et à pouvoir communiquer avec les autres modules.

2.3.5 Gestion de la variabilité

Tous les produits de la SPL n'auront pas les mêmes caractéristiques, c'est le point fort de la personnalisation de masse. Au sein d'une SPL, il existe donc une notion de variabilité entre les produits. La gestion de la variabilité se fait aussi bien au niveau du domain engineering que de l'application engineering (voir Section 2.4). Dans l'exemple, la variabilité est exprimée par le fait que chaque produit possède sa propre combinaison de modules. Pour plus d'informations, le Chapitre 3 est consacré à la variabilité.

Terminologie utilisée

A partir de maintenant, la terminologie utilisée va être simplifiée afin de faciliter la compréhension du mémoire.

Les termes "sofware product lines" (SPLs) et "product lines" (PLs) seront considérés comme équivalent. Ils pourront désigner :
- Une entité physique imaginaire, une "chaîne de montage logiciel". Par exemple : "Une SPL est capable de produire des logiciels très bon marchés".
- L'ensemble des produits développables à partir des artéfacts de la plateforme.

De même, les termes "produit", "logiciel", "application" et "software" seront aussi considérés comme équivalents. Ils représenteront les ouputs de la SPL, développés à partir des artéfacts.

2.4 Les activités-clés d'une SPL

Pour une entreprise ayant peu de connaissances dans le domaine des SPLs, en développer une peut se révéler être une tâche ardue. Heureusement, avec le temps, des scientifiques et des industriels ont synthétisé les connaissances indispensables à leur création. Cette section a pour but de présenter de façon générique les processus nécessaires à la création d'une SPL.

L'exemple imagé de la Figure 2.3 (située sur la page suivante) permet de donner une vue globale d'une SPL. Une SPL est donc composée de deux grands processus, le domain engineering et l'application engineering. Le premier sert à créer la PL elle-même tandis que le second permet de développer les produits. Chacun de ces deux processus est bien sûr exécuté en respectant les principes-clés des SPLs introduits dans la Section 2.3. Cependant, le succès d'une SPL ne dépend pas que des processus, il faut aussi être capable de gérer efficacement toute la ligne de produits. Les sous-sections suivantes vont permettre d'expliquer chacune de ces activités en détail.

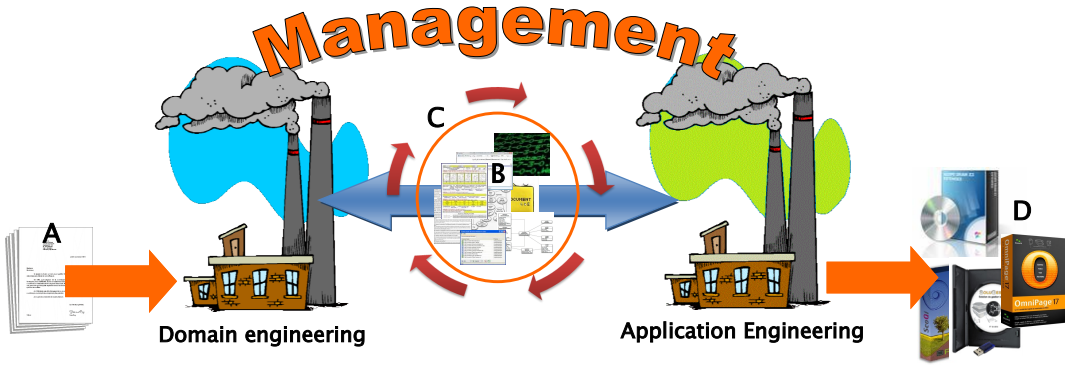

FIGURE 2.3 – Représentation imagée d'une SPL.

Dans l'exemple de la Figure 2.3, le point A représente les inputs dont les ingénieurs domaine ont besoin pour produire les artéfacts (point B) qui formeront la plate-forme (point C). A leur tour, les ingénieurs software utiliseront ces artéfacts pour développer les différentes applications (point D) de la SPL. Le cercle de flèches rouges illustre le développement continu des artéfacts tandis que la double flèche bleue représente les interactions qui existent entre domain engineering et application engineering.

Dans le contexte des SPLs, les ingénieurs ou développeurs domaine sont les professionnels employés durant le domain engineering tandis que les ingénieurs ou développeurs software sont les professionnels employés durant l'application engineering.

2.4.1 Domain engineering

Comme expliqué un peu plus haut, le domain engineering permet de concevoir la SPL elle-même, c'est-à-dire la plate-forme (C sur la Figure 2.3) et ses artéfacts (B sur la Figure 2.3). Pour cela, un certain nombre d'inputs (A sur la Figure 2.3) vont être nécessaires. Ces derniers peuvent varier en fonction de l'organisation, du framework utilisé, ... En général, ils seront par exemple composés de [10] [22] :

- La liste des produits que la SPL devra être capable de produire et les exigences liées à ceux-ci.
- Les contraintes de production. Ces dernières pourront imposer certains standards ou exigences que la SPL devra respecter.
- La stratégie de production, qui décrit de manière globale la façon dont la SPL devra être construite. Elle pourra aussi indiquer les modes de production des artéfacts (s'ils pourront être achetés, récupérés,...).

Une fois tous ces éléments obtenus, les ingénieurs domaine vont débuter la conception de la plate-forme en "réunissant" (en développant de nouveaux artéfacts, en récupérant d'anciens artéfacts, ...) tous ses artéfacts. Les développeurs domaine vont d'abord commencer par déterminer les exigences communes à tous les produits de la SPL et celles liées à des produits spécifiques. A partir de là, ils vont pouvoir commencer à **modéliser la variabilité** (voir Chapitre 3) et à

produire les différents artéfacts qui permettront de construire les produits. Ces artéfacts devront respecter la variabilité précédemment décrite, au risque de ne pas pouvoir être réutilisables. Lors du domain engineering, différents outputs sont produits. Ci-dessous, en voici quelques exemples [10] [22] :

- **Le modèle de variabilité**. Ce dernier permet de représenter la variabilité de la SPL. Dans le chapitre suivant, un certain type de modèle de variabilité est présenté : **les diagrammes de features**.
- L'architecture de référence. Elle représente la structure globale de la SPL et englobe la structure valide de chacun des produits.
- Le plan de production. Il décrit comment les artéfacts devront être assemblés entre eux pour produire les applications.

L'activité préliminaire du développement des artéfacts se terminera quand tous les outputs auront été produits. Cependant, il ne faut pas croire que le domain engineering se limite à cela. Les interactions avec l'application engineering (voir Sous-section 2.4.3) et l'ajout de nouveaux produits obligent les ingénieurs domaine à développer et à maintenir les artéfacts en continu.

2.4.2 Application engineering

Les ingénieurs software réutilisent les artéfacts créés lors du domain engineering pour développer les différents produits (D sur la Figure 2.3) de la SPL. Dans la plupart des cas, si la SPL a bien été conçue (vue à long terme, flexibilité des artéfacts,...), les programmeurs software vont tout d'abord "calculer" un delta entre les fonctionnalités de l'application à développer et celles de la SPL. Grâce à cela, ils vont pouvoir détecter les artéfacts de la plate-forme qu'il sera possible d'utiliser. A partir de ce moment, s'il n'est pas nécessaire de développer des fonctionnalités spécifiques au produit, ils pourront utiliser le plan de production pour assembler les artéfacts sélectionnés entre eux et les paramétrer si nécessaire. Ils finiront ainsi par produire le logiciel commandé. Lors de l'application engineering, une des **activités-clés** est la **fixation** ("binding") **de la variabilité** (voir Chapitre 3). Durant l'application engineering, différents outputs intermédiaires sont produits. Ci-dessous, en voici quelques exemples [10] [22] :

- L'instantiation de l'architecture de référence en fonction des exigences du produit.
- **L'instantiation du modèle de variabilité**. Les développeurs ont fixé la variabilité en faisant les meilleurs choix possibles, selon les fonctionnalités requises pour le logiciel (voir chapitre suivant pour plus de détails).
- Des composants logiciels paramétrés pour les besoins de l'application.

2.4.3 Interactions continues et maintenance de la plate-forme

Les deux activités-clés décrites ci-dessus ont été présentées indépendamment l'une de l'autre. Pourtant, dans la réalité, Domain engineering et Application engineering sont intimement liés et ils ne cessent d'interagir l'un avec l'autre. La double flèche bleue de la Figure 2.3 illustre ces échanges. Les données échangées peuvent être de différents formes : feed-back (les ingénieurs software peuvent

par exemple prévenir les développeurs domaine qu'un artéfact ne fonctionne pas correctement) , artéfacts,...

Parallèlement à ces interactions, les artéfacts sont développés et maintenus en continu. Leur développement ne se limite donc pas à la phase initiale de création de la SPL. Le cercle de flèches rouges de la Figure 2.3 illustre ce processus continu. Les artéfacts doivent par exemple évoluer. Si la direction décide d'agrandir la gamme de produits de la SPL, les développeurs domaine seront obligés d'adapter les artéfacts existants. Et si nécessaire, ils seront aussi amenés à en développer de nouveaux qui répondront aux nouvelles exigences. Les artéfacts doivent aussi être maintenus. Par exemple, en fonction des avancées technologiques (nouveaux systèmes d'exploitation, ...), il faudra parfois mettre la plate-forme à jour afin que les produits puissent eux-mêmes être maintenus à jour et rester compétitifs.

2.4.4 Management

Le succès d'une SPL repose aussi sur sa gestion. Il faut s'assurer que toutes les activités sont bien coordonnées et qu'elles reçoivent les ressources nécessaires. Une gestion efficace est un des facteurs de succès les plus importants pour une SPL. Les activités de management d'une SPL consistent par exemple à s'assurer que le personnel du domain engineering et de l'application engineering est engagé dans les bonnes activités et qu'il suit les processus définis. De manière similaire, il faut aussi contrôler que des risques (mauvaise communication, manque de ressources,...) ne viennent pas compromettre le succès de la SPL. Dans [10], Clements et Northrop définissent le management comme étant "*[...] the authority that is responsible for the ultimate success or failure of the product line effort*".

2.5 Avantages et facteurs d'échec des SPLs

Dans cette section, la première sous-section va permettre d'exposer quelques avantages liés à l'utilisation d'une SPL. Ensuite, la deuxième sous-section listera les principaux facteurs d'échec d'une ligne de produits.

2.5.1 Avantages des SPLs

Pour une société, le développement et l'utilisation efficace d'une SPL présentent de nombreux avantages. Ci-dessous, en voici quelques exemples [22].

Réduction des coûts de développement

Au départ, la création d'une SPL nécessite de nombreux investissements ("upfront investments"), il faut définir l'ensemble des produits qui pourront être créés, développer les artéfacts, définir les plans pour assembler ces artéfacts, ... Et pendant ce temps, aucun produit n'est vendu. Cependant, quand la SPL sera prête et à partir de trois ou quatre produits ("break-even point"), le coût de développement de chaque logiciel sera bien plus faible que s'il avait été conçu de manière classique. En effet, la réutilisation des artéfacts permet de réduire le coût de développement de chaque produit. La Figure 2.4 (située sur la page suivante) présente un graphique permettant d'illustrer ce phénomène.

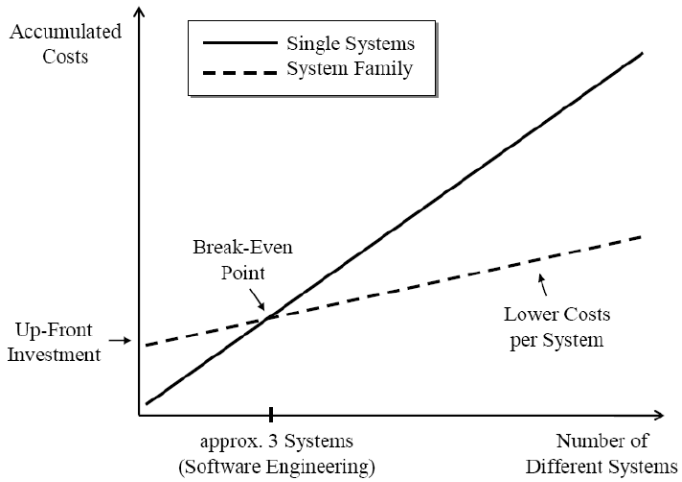

FIGURE 2.4 – Coûts de développement des produits d'une SPL comparés aux coûts de développement de systèmes unitaires [22].

Sur ce graphique, le trait continu représente l'accumulation des frais de développement de systèmes créés de façon classique (sans l'aide d'une SPL). Pour chaque système, le coût est considéré comme constant. Le trait en pointillés représente l'accumulation des frais de développement de systèmes créés à l'aide d'une SPL. Le point de rencontre entre ces deux traits est le "break-event point", il représente le nombre de produits (approximativement trois) à partir duquel il devient plus rentable d'utiliser une SPL pour les produire.

Augmentation de la qualité générale des produits

Lors de la création d'une SPL, chaque artéfact est soumis à des plans de test rigoureux (normalement, comme pour tout programme classique). Cependant, grâce aux SPLs, l'assurance de la qualité de l'artéfact peut encore être améliorée. En effet, plus un artéfact sera réutilisé, plus il sera testé, ce qui augmente fortement la possibilité de dépister un bug. De plus, si un défaut devait quand même être détecté, l'artéfact coupable serait modifié et la qualité des produits ne ferait encore qu'augmenter.

Réduction des délais de commercialisation

Grâce à une SPL, les délais de commercialisation des produits peuvent être réduits. Il faut cependant attendre la fin de la période de développement des artéfacts pour pouvoir commercialiser le premier produit. Néanmoins, à partir de ce moment, grâce à la réutilisation des artéfacts, une SPL sera en mesure de développer les produits de plus en plus rapidement. Le graphique de la Figure 2.5 (située sur la page suivante) permet d'illustrer ce phénomène.

Sur ce graphique, le trait continu représente les délais de commercialisation de systèmes développés de manière traditionnelle. Pour chaque système, ce délai est considéré comme constant. Le trait en pointillés représente les délais de commercialisation de systèmes développés grâce à une SPL. Pour chaque produit, ce délai est de plus en plus en court. Le point de croisement entre ces deux traits

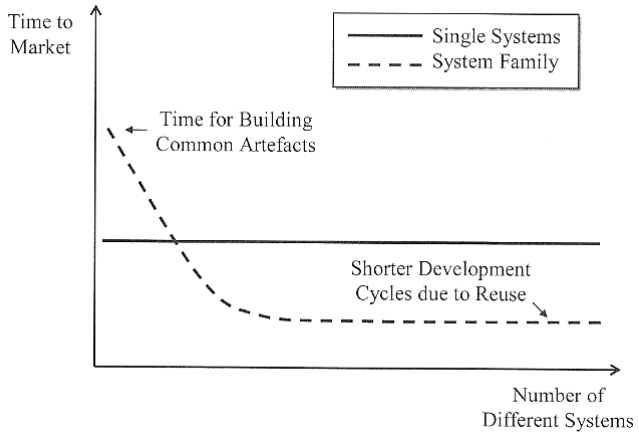

FIGURE 2.5 – Comparaison des délais de commercialisation des produits d'une SPL par rapport à ceux de systèmes unitaires [22].

représente le nombre de produits à partir duquel une SPL permet de produire les logiciels plus rapidement.

Gestion plus aisée de la complexité

Ces dernières années, la complexité des logiciels a considérablement augmenté. Un programme peut maintenant faire des dizaines de millions de lignes de code. Gérer la complexité de gros programmes est devenu un problème majeur pour les développeurs. Grâce à leurs plate-formes, les SPLs permettent de réduire la complexité. La gestion de la variabilité et la réutilisation efficace des artéfacts, selon des plans et des processus bien précis, simplifient la conception des programmes.

2.5.2 Facteurs d'échec des SPLs

Comme expliqué dans la sous-section précédente, l'utilisation d'une SPL présente plusieurs avantages. Cependant, créer et développer une PL demande beaucoup d'efforts. De nombreux facteurs peuvent venir compromettre ce processus. Ci-dessous, voici cinq causes connues d'échec d'une SPL :

Manque de liquidité pour les investissements front-up

La création d'une SPL demande de nombreux investissements, notamment pour le développement de tous les artéfacts (voir Sous-section 2.5.1). Durant cette période, la société doit être capable de fournir les ressources nécessaires à la réalisation des différentes phases de développement. Si cela n'est pas le cas, la SPL ne pourra tout simplement pas être développée et elle sera donc abandonnée. De plus, même si la compagnie a été capable de fournir ces ressources, elle devra peut-être aussi faire face financièrement à une période durant laquelle les bénéfices dégagés seront inférieurs aux coûts de développement des produits. Ce ne sera qu'à partir d'un certain nombre de produits que la société pourra

commencer à engranger des bénéfices. Avant de se lancer dans le développement d'une SPL, une entreprise doit donc s'assurer d'avoir les ressources nécessaires pour cela.

Produits trop peu nombreux

Comme précisé ci-dessus, une société ne pourra dégager des bénéfices d'une SPL qu'à partir d'un certain nombre de produits développés. Si une société développe une SPL en prévoyant trop peu de produits, les investissements de départ et les coûts de développement pourraient toujours rester supérieurs aux recettes. La SPL ne pourrait donc provoquer que des pertes en ne présentant aucun des avantages de la sous-section précédente.

Produits trop diversifiés

Si la famille de produits est trop diversifiée, le travail demandé pour développer des artéfacts assez flexibles pourrait provoquer des coûts supplémentaires. De plus, le nombre d'artéfacts réutilisables sera sans doute faible, ce qui fera perdre les avantages liés à leur réutilisation. Au final, pour chaque produit, son coût de développement pourrait devenir supérieur au coût qu'il aurait fallu pour le développer de manière traditionnelle (sans l'aide d'une SPL). Encore une fois, la SPL ne pourrait donc dégager que des pertes.

Connaissance insuffisante du domaine

Créer une SPL et développer ses produits nécessitent une excellente connaissance de son domaine (technologies, marché, clients, ...). La compagnie doit tout d'abord être capable de correctement déterminer l'ensemble des produits en n'oubliant pas de prendre en compte les besoins futurs des clients. Dans le cas contraire, si les managers n'ont pas été capables de prévoir tous les produits, il faudra modifier la SPL pour y ajouter de nouveaux produits, ce qui engendrera des frais supplémentaires. Au niveau du domain engineering, les ingénieurs domaine devront aussi avoir une bonne connaissance du domaine afin de correctement modéliser la variabilité (voir Chapitre 3). Lors de l'application engineering, afin d'efficacement fixer la variabilité, les développeurs software devront aussi connaître le domaine. De mauvais choix pourraient entraîner la création de produits qui ne répondront pas aux exigences requises. Dans une SPL, chaque acteur doit donc connaître le domaine, au risque de provoquer l'échec de la PL.

Instabilité du domaine

Si le domaine auquel la SPL s'attache est instable et qu'il évolue trop rapidement (par exemple, tous les six mois), les différents investissements nécessaires au maintien de la SPL risquent de ne jamais pouvoir être rentabilisés. En effet, pour chaque changement, il faudrait adapter les artéfacts et les bénéfices dégagés grâce aux produits risqueraient de ne jamais pouvoir couvrir les coûts de toutes ces adaptations.

Chapitre 3

La variabilité au sein des SPLs : concepts et modélisation

Le chapitre précédent a permis de présenter le contexte globale de la problématique : les SPLs. Dans le présent chapitre, nous allons affiner ce contexte en nous attardant sur un des concepts-clés (voir Section 2.3) des lignes de produits : la variabilité. La gestion et la modélisation de cette dernière représente une tâche critique au sein d'une SPL. A ce niveau, un échec pourrait anéantir toute chance de succès de la ligne de produits. La première section sera consacrée à la variabilité elle-même et à tous les concepts qui l'entourent : les différents états de la variabilité, le binding time, ... Ensuite, la seconde section s'intéressera à la modélisation de la variabilité et plus précisément, à un certain type de modélisation : les diagrammes de features. La première technique de modélisation à être présentée sera FODA [16]. Par après, certaines de ses extensions, en accord avec la technique de modélisation introduite dans le chapitre suivant, seront elles aussi décrites à leur tour.

3.1 La variabilité au sein des SPLs

Comme expliqué au chapitre précédent, une SPL est composée de plusieurs produits. Ces derniers partagent des caractéristiques communes mais chacun d'entre eux possède aussi ses propres particularités. Au sein d'une SPL, entre les différents produits, il existe donc une notion de variabilité. Dans l'exemple des PDA de la Figure 2.2 du chapitre précédent, les trois systèmes d'exploitation (B100, B200, B300) ont en commun le module de base. De manière similaire, les OS B200 et B300 partagent les modules Bluetooth et 3G. Enfin, le module WiFi est une particularité de l'OS B300.

Dans un artéfact, la variabilité est souvent représentée par des **points de variation** [1] ("Variation Point") [22]. Chaque point de variation représente une caractéristique, une propriété, ... (de l'artéfact) qui varie en fonction du pro-

1. Le mécanisme du point de variation est générique. Il sera souvent remplacé par un mécanisme semblable plus adapté au modèle de variabilité utilisé.

duit dans lequel l'artéfact est employé. Les différentes instances possibles d'un point de variation sont appelées les **variants** ("Variants") [22]. Selon le produit développé, il faudra donc sélectionner un certain variant du point de variation. Dans l'exemple de la Figure 2.2, on pourrait imaginer un point de variation au niveau de l'artéfact du module Bluetooth. Ce point de variation correspondrait au driver de l'antenne Bluetooth du PDA. Il y aurait deux variants : un premier correspondant au driver de l'antenne du B200 et un second correspondant à l'antenne du B300. Il suffirait de choisir le driver (variant) en fonction du modèle de PDA sur lequel l'OS sera utilisé. La Figure 3.1 illustre cet exemple. La variabilité introduite dans cet artéfact permet donc de le réutiliser dans deux logiciels différents.

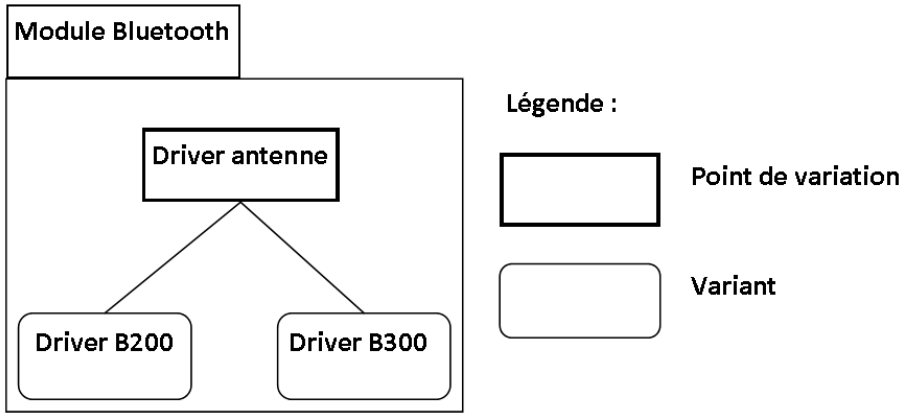

FIGURE 3.1 – Point de variation et variants.

3.1.1 Variabilité dans le temps et variabilité dans l'espace

Dans [22], Pohl, Böckle et Linden présentent les différences qui existent entre la variabilité dans le temps et la variabilité dans l'espace. Au cours de l'utilisation d'une SPL, un artéfact ne va cesser d'évoluer (voir Sous-section 2.4.3), entre autres à cause de l'utilisation de nouvelles technologies, l'introduction de nouvelles lois, ... Au cours du temps, il pourra donc exister différentes versions d'un artéfact. C'est ce qu'on appelle l'évolution ou la variabilité dans le temps. Différents produits d'une SPL pourront donc avoir été construits avec différentes versions d'un même artéfact. Afin de pouvoir gérer correctement l'évolution des différents artéfacts, il est nécessaire de mettre en place une stratégie de gestion des configurations. De plus, cette dernière permettra de toujours garder la trace des différentes versions des artéfacts utilisés. La Figure 3.2 (située sur la page suivante) permet d'illustrer le principe de la variabilité dans le temps. Dans cet exemple, nous pouvons voir qu'il a existé trois versions différentes de l'artéfact X. Au fur et à mesure de ces versions, le point de variation (PV) a compté de plus en plus de variants.

Il est aussi intéressant de noter que dans les lignes de produits, l'évolution des artéfacts est souvent facilitée grâce au mécanisme des points de variation. Il est souvent plus facile d'introduire des changements au niveau de ces points. Dans le cas de modifications prévues, il ne sera donc parfois nécessaire "que"

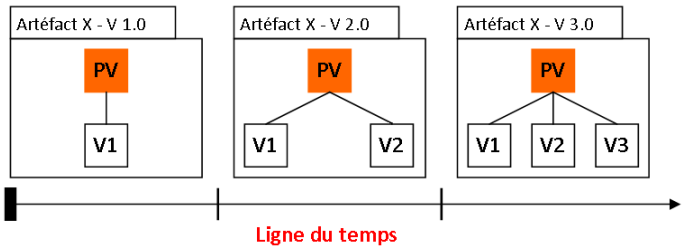

FIGURE 3.2 – Variabilité dans le temps d'un artéfact.

d'ajouter des variants à certains points de variation. Dans le cas contraire, les changements pourraient nécessiter de plus gros travaux : reconstruction de l'artéfact, introduction de nouveaux points de variation, ...

Parallèlement à la variabilité dans le temps, il existe donc une variabilité dans l'espace. Pour un artéfact, elle représente les différentes "formes" que ce dernier peut avoir à un moment précis. Ici, une "forme" représente un artéfact où les points de variation ont été fixés grâce à la sélection de certains variants. Une forme d'un artéfact sera donc destinée à un ou plusieurs produits spécifiques. La Figure 3.3 explicite le principe de la variabilité dans l'espace.

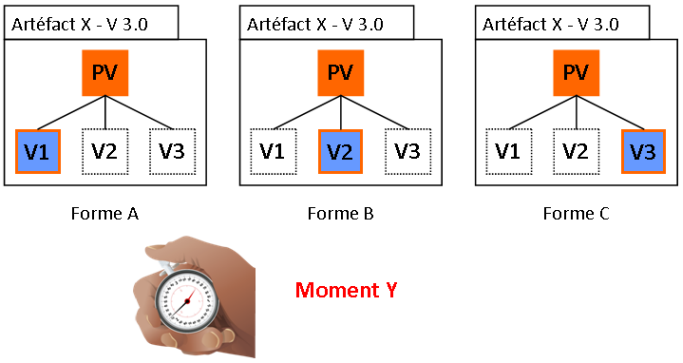

FIGURE 3.3 – Variabilité dans l'espace d'un artéfact.

Si nous reprenons la version 3.0 de l'artéfact X de la Figure 3.2 et si nous arrêtons le temps à un moment précis Y, l'artéfact X pourra avoir avoir trois formes. La forme A où le premier variant (V1) a été sélectionné, la forme B où le deuxième variant (V2) a été choisi et enfin, la forme C où le dernier variant (V3) a été préféré.

La variabilité dans le temps représente donc toutes les versions qu'il a pu exister d'un même artéfact au cours de la vie d'une SPL. De son côté, **la variabilité dans l'espace** représente les différentes formes qu'un artéfact peut adopter à un moment précis. **Dans le cadre de ce mémoire, nous nous concentrerons sur cette dernière**. C'est cette variabilité, spécifique aux SPLs, qui nécessite une gestion toute particulière. La variabilité dans le temps existe aussi dans le développement de logiciels classiques. Avec quelques adaptations, les processus de gestion des configurations classiques deviennent parfaitement capables de traiter aussi la variabilité dans le temps des SPLs.

3.1.2 Niveau d'abstraction

Lors du développement d'un produit, des artéfacts se situant à différents niveaux d'abstraction (architecture, exigences, ...) sont utilisés. Dans leur article [30], Gurp et al. expliquent que la variabilité, à travers ces artéfacts, peut elle-même être introduite à différents niveaux d'abstraction. Certains points de variation seront, par exemple, introduits au niveau de l'architecture de référence tandis que d'autres seront introduits dans des modules, des plans de tests, ... Le choix d'introduire certains points de variation à certains niveaux d'abstraction est le résultat de décisions prises par les ingénieurs domaine. A un certain niveau d'abstraction, la variabilité peut se retrouver dans trois états différents :

1. Implicite ("Implicit") : La variabilité (les points de variation) a été introduite à un certain niveau d'abstraction. Ceci veut dire que cette variabilité est aussi présente aux niveaux d'abstraction supérieurs, mais de façon implicite.
2. Désignée ("Designed") : La variabilité est considérée comme désignée au niveau d'abstraction où ses points de variation ont été introduits, elle est explicite.
3. Fixée ("Bound") : La variabilité est fixée quand tous les points de variation la représentant ont eux-mêmes été fixés, c'est-à-dire, quand les ingénieurs domaine ont sélectionné le ou les variants nécessaires pour chacun de ces points de variation.

3.1.3 Variabilité interne et externe

Dans [22], les auteurs différencient deux types de variabilité : la variabilité interne et la variabilité externe. Cette distinction permet de masquer une partie de la variabilité aux clients. En effet, ces derniers ne seront conscients que de la variabilité externe tandis qu'ils ignoreront tout de la variabilité interne. Cela permet de ne présenter aux clients que de la variabilité qui a un sens à leurs yeux tout en augmentant leur satisfaction (ils peuvent choisir les variants qui répondent le mieux à leurs exigences). De plus, cela permet aussi aux ingénieurs logiciel de garder la main mise sur certaines décisions plus techniques (dans l'exemple de la Figure 3.1, le choix du bon driver en fonction du modèle de PDA). Souvent, les clients pourront fixer la variabilité externe de deux façons :

1. Directement : pour chaque point de variation, le client décide quel(s) variant(s) choisir.
2. Indirectement : pour chaque point de variation, les ingénieurs/commerciaux pré-sélectionnent un ensemble de variants. Ils définissent donc un sous-ensemble de produits de la SPL parmi lesquels les clients pourront choisir.

Souvent, la variabilité interne est une conséquence de la variabilité externe. Les décisions prises (les variants sélectionnés) par les clients aux niveaux d'abstraction les plus élevés nécessitent souvent d'être affinées dans des niveaux d'abstraction moins élevés. Comme montré un peu plus haut, le client ne sera intéressé que par certaines décisions. Les différentes alternatives de réalisation de ces décisions ne l'intéresseront donc pas. Elles seront alors considérées comme de la variabilité interne. De plus, la variabilité interne pourra aussi être le résultat du raffinement de la variabilité interne située à un niveau d'abstraction plus élevé. Enfin, certaines raisons plus techniques (choix d'un module logiciel, choix d'un

protocole,...) pourront pousser les ingénieurs domaine à ajouter eux-mêmes de la variabilité interne.

Tant la variabilité externe que la variabilité interne peuvent donc avoir pour conséquence commune d'introduire encore plus de variabilité interne, ce qui a donc comme résultat d'augmenter de plus en plus la complexité de la variabilité au fur et à mesure des niveaux d'abstraction. La pyramide de la Figure 3.4 illustre ce phénomène.

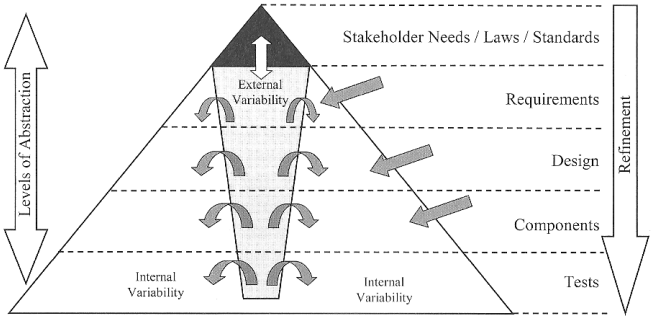

FIGURE 3.4 – Pyramide de la variabilité [22].

Au sommet de la pyramide se trouvent les exigences, les lois et les standards que les applications de la SPL devront respecter. A partir de là, les ingénieurs domaine créent les premiers artéfacts au niveau d'abstraction des exigences. Ces derniers contiennent souvent plus de variabilité externe (zone grisée) que de variabilité interne (zone blanche). Le raffinement (flèches incurvées) de cette variabilité externe va avoir comme conséquence d'augmenter la variabilité interne, aussi bien au niveau d'abstraction actuel qu'au niveau suivant. De plus, le raffinement de la variabilité interne du niveau actuel va aussi avoir comme résultat d'augmenter la variabilité interne du niveau suivant. Enfin, l'introduction supplémentaire (flèches partant de l'extérieur et pointant l'intérieur de la pyramide) de variabilité par les ingénieurs va encore augmenter cette dernière. Ce phénomène va se reproduire de niveau en niveau, plus on descendra dans la pyramide, plus la variabilité interne et la complexité augmenteront. A l'inverse, au fur et à mesure des niveaux, la variabilité externe diminuera. Ceci est dû au fait que les décisions concernant les clients se situeront souvent aux niveaux d'abstraction les plus élevés. La forme de la pyramide témoigne donc de cette augmentation de la complexité au travers des différents niveaux d'abstraction. Il est donc nécessaire de gérer correctement et efficacement cette dernière. Dans le cas contraire, la SPL pourrait être un échec. Il faut aussi noter que l'augmentation de la complexité n'est pas seulement due à l'augmentation de la variabilité, elle est aussi due à l'augmentation des relations entre tous les points de variation, les variants et les artéfacts de la SPL.

3.1.4 Mécanismes de variation

La variabilité est donc incarnée par des points de variation introduits à différents niveaux d'abstraction. Les points de variation permettent de modéliser la variabilité de façon générique. Ces points sont eux-mêmes implémentés "phy-

siquement" grâce à des mécanismes de variation. A titre informatif, voici quatre exemples de mécanismes de variation [4] : la paramétrisation (un artéfact peut par exemple être personnalisé grâce à certains paramètres), la génération (un outil prend comme entrée une liste de spécifications et produit l'artéfact personnalisé voulu), la configuration (un outil assemble un ensemble de blocs de code - qui sont eux-mêmes des artéfacts - pour produire un composant spécifique à un ou plusieurs produits) et l'héritage (dans les logiciels orientés objet, il permet par exemple d'introduire dans un artéfact une classe nécessaire à un certain produit).

Lors de la sélection d'un mécanisme de variation, les ingénieurs domaine doivent prendre de nombreux facteurs en compte [4] :

- Les mécanismes de variation disponibles : certains mécanismes sont adaptés à des artéfacts situés à des niveaux d'abstraction déterminés, certains mécanismes nécessitent telle technologie ou telle compétence pour être implémentés, ... Par exemple, le mécanisme de l'héritage des classes demandera un développeur ayant des connaissances en programmation orientée objet.
- Des informations sur les produits : ces informations pourront par exemple être des documents (diagrammes de features, ...) décrivant les différences entre tous les produits de la SPL.
- La stratégie de production : cette dernière pourra imposer certaines contraintes (budget, temps, ...). Les ingénieurs domaine devront donc sélectionner des mécanismes permettant de respecter ces contraintes. La stratégie pourra aussi informer les ingénieurs domaine des connaissances des ingénieurs logiciel : si ces derniers n'ont pas les aptitudes nécessaires pour utiliser un certain mécanisme de variation, il ne sert à rien de l'implémenter. A côté de cela, la stratégie de production fournit aussi d'autres renseignements (niveau de compétence des clients, des utilisateurs, ...) utiles aux ingénieurs domaine.
- ...

Durant le domain engineering, si les ingénieurs domaine ne disposent pas de toutes ces informations, ils pourront être forcés de sélectionner les mécanismes de variation à "l'aveugle", en essayant de faire au mieux. Dans ce cas, les mécanismes de variation risquent de déterminer eux-mêmes les produits et la stratégie de production. Ceci pourrait donc conduire à une SPL ne respectant pas (tous) les objectifs de l'entreprise.

3.1.5 Le binding time

Comme expliqué précédemment, un point de variation possède un certain nombre de variants. A un moment, les acteurs (ingénieurs software, clients, utilisateurs, ...) vont pouvoir décider de fixer ce point en sélectionnant un de ses variants. Cependant, le moment où un point est fixé, le binding time, n'est pas anodin. En effet, les ingénieurs domaine décident à l'avance, à quelle(s) étape(s) du développement ou de l'utilisation du produit, un point doit être fixé. Ils doivent donc trouver le moment et le mécanisme de variation (voir sous-section précédente) optimaux pour chaque point de variation. Un tel point pourra par exemple être fixé [2] :

- Durant l'analyse des exigences des clients : sur un diagramme de features

(voir section suivante), les clients pourront choisir de sélectionner certains éléments.
- Durant le développement du produit : les ingénieurs software pourront choisir de fixer un point à l'aide d'un variant précis.
- Au moment de l'installation du logiciel : en fonction du système d'exploitation, le logiciel pourra choisir d'installer une librairie spécifique.
- Durant l'exécution du logiciel : un utilisateur pourra par exemple sélectionner une langue parmi toutes celles proposées par le logiciel.
- ...

Les binding times ont aussi pour rôle d'assurer la cohérence des décisions, c'est-à-dire, de garantir que toutes les décisions concernant la fixation des points de variation ont bien été prises dans un ordre logique. Il faut aussi signaler que plus les binding times seront définis tardivement, plus le logiciel développé sera flexible. Le programme ne deviendra "spécifique" que tard dans le développement. Certaines étapes du développement pourront donc devenir identiques pour plusieurs produits car elles ne contiendront pas de points de variation, il ne sera pas nécessaire de les répéter pour chaque produit. Souvent, les binding times de systèmes embarqués seront définis tôt dans le développement. En effet, la configuration du système sur lequel le logiciel sera exécuté est connue à l'avance. Dans le cas de systèmes non embarqués, la configuration ne sera pas connue à l'avance. Il pourra donc être intéressant de définir tardivement certains binding times. De cette manière, l'utilisateur pourra lui-même personnaliser le logiciel en fonction de sa configuration personnelle. Afin de pouvoir définir les binding times des points de variation, les ingénieurs domaine doivent donc prendre en compte un nombre important de paramètres (type du logiciel développé, mécanismes de variation existants, type de variabilité représentée, état de développement du logiciel, ...).

3.2 Modélisation de la variabilité

Comme expliqué dans la Sous-section 3.1.3, la quantité de variabilité augmente au fur et à mesure des niveaux d'abstraction. Dans certaines lignes de produits, la complexité de la variabilité peut être très élevée. Si cette dernière n'est pas gérée efficacement, elle peut mettre en péril le succès de la SPL. La modélisation de la variabilité permet d'assister les ingénieurs domaine dans la gestion de la complexité. Aujourd'hui, après plusieurs années de recherches, les scientifiques ont mis au point différentes méthodes permettant de modéliser la variabilité. Chacune de ces méthodes possède certaines particularités (concepts, mécanismes de modélisation, niveau d'abstraction, ...) qui font qu'elles sont plus adaptées à certains types de SPLs. Malheureusement, les entreprises n'ont pas toujours conscience de la tâche critique que représente la modélisation de la variabilité. Il arrive que ce travail soit considéré comme une activité annexe et que la variabilité elle-même soit introduite tardivement (au moment du design ou de l'implémentation) dans les artéfacts [28]. Afin de s'assurer de la réussite de leur SPL, il est important que les entreprises tiennent compte de la variabilité et apprennent à la gérer durant toute la durée de vie de la SPL.

En accord avec la technique de modélisation présentée dans le chapitre suivant, dans cette section, nous nous attarderons surtout sur un certain type de

modèle de représentation de la variabilité : les diagrammes de features. Dans [16], Khang et al. sont les premiers à avoir présenté une technique de modélisation basée sur le concept de feature : FODA. Par la suite, de nombreux scientifiques ont développé des extensions destinées à améliorer cette technique. La sous-section suivante sera donc consacrée aux diagrammes de features. Enfin, la seconde sous-section présentera très brièvement d'autres types de techniques de modélisation de la variabilité.

3.2.1 Techniques de modélisation basées sur les features

La technique de modélisation exposée dans le chapitre suivant réutilise un certain nombre d'extensions introduites afin d'améliorer FODA. Dans cette sous-section, nous nous limiterons donc à la présentation de ces extensions (en plus de la présentation de FODA). Dans notre cas, les points importants à retenir seront les nouvelles "possibilités sémantiques" (ce qu'elles permettent d'exprimer et que FODA ou des extensions antérieures ne permettaient pas d'exprimer) offertes par ces extensions. Même si la syntaxe des ces extensions sera brièvement présentée, là n'est pas l'essentiel. Souvent, il arrive que des auteurs réutilisent, de manière plus ou moins implicite, des extensions précédemment introduites en les redéfinissant à l'aide de leur propre syntaxe. Le plus important reste donc la sémantique.

Cependant, avant de passer à la présentation de FODA, il est important de tout d'abord définir le concept de "feature". Dans le domaine des SPLs, de nombreux scientifiques ont tenté d'en donner leur propre définition (voir [28] et [3] pour différents exemples de définitions). Ici, nous adopterons la définition de référence, c'est-à-dire, la définition contenue dans le rapport technique de FODA [16] : "*the attributes of a system that directly affect end-users*" ou encore "*A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems*". Les diagrammes de features sont des modèles de haut niveau qui doivent être facilement compréhensibles par tous les acteurs, notamment par les personnes n'ayant pas ou peu de connaissances en informatique.

Ce genre de modèle est surtout destiné à pouvoir établir un dialogue, à un haut niveau d'abstraction, entre les différents acteurs (développeurs, clients, ...) de la SPL. Par exemple, un tel modèle pourra servir de moyen de communication entre les clients et les développeurs. A ce moment-là, il faudra présenter un diagramme contenant uniquement les features qui possèdent un intérêt aux yeux des clients (variabilité externe). Toutes les autres features (variabilité interne) devront être temporairement écartées. De manière similaire, un modèle de features pourra aussi être utilisé par les développeurs pour communiquer entre eux. Dans ce cas, les features pourront représenter des éléments plus techniques tels que des protocoles de communication, des formats de données, ... Les objets ou caractéristiques représentés par les features doivent être clairs et avoir du sens aux yeux des acteurs. Une feature au sens ambigu (dont l'objet ou la caractéristique qu'elle représente est ambigu) pourra par exemple tromper un acteur et lui faire prendre de mauvaises décisions. Le produit développé pourrait alors ne pas répondre aux exigences du client.

FODA

En 1990, Kang et al. publient un rapport technique [16] présentant une nouvelle technique de modélisation : FODA. Cette dernière permet de détecter et de modéliser les points communs et les différences entre différents logiciels. La méthode elle-même est divisée en trois phases successives. La première, l'analyse du domaine, permet de définir le champ d'application de la SPL qui lui-même conduit à la construction de la liste des produits développables. La deuxième, la modélisation du domaine, est consacrée à l'analyse fonctionnelle, la modélisation des features et des relations entre ces dernières (diagramme de features). La dernière, la modélisation de l'architecture, permet de finalement définir l'architecture de la SPL.

En accord avec l'objectif de la sous-section parente, nous nous focaliserons sur les diagrammes de features construits lors de la seconde phase. Un tel diagramme consiste en un arbre dont la racine incarne un certain concept (un logiciel de traitement de texte, un système d'exploitation, ...). Chaque noeud de cet arbre représente une feature ou une sous-feature (qui elle-même est une feature qui peut aussi posséder des sous-features). La relation de base entre une feature et ses sous-features est de type "consiste en" ("*consists-of*"). Grâce à certains mécanismes, FODA permet de personnaliser leur sélection :

- Feature obligatoire : Si la feature parente d'une sous-feature obligatoire est sélectionnée, cette sous-feature doit aussi être sélectionnée.
- Feature optionnelle : Même si la feature parente d'une sous-feature optionnelle est sélectionnée, cette sous-feature ne devra pas forcément aussi être sélectionnée, cela dépendra des choix des acteurs. Une feature optionnelle est représentée grâce à un cercle vide au-dessus de son nom.
- Features alternatives : Une sous-feature alternative est considérée comme une spécialisation de sa feature parente. Dans un groupe de sous-features alternatives, une et une seule sous-feature peut être sélectionnée (équivalent de l'opérateur logique "XOR"). Graphiquement, les branches (partant de la feature parente vers chaque sous-feature) d'un groupe de sous-features alternatives sont reliées par un arc. Dans FODA, un point de variation (voir section précédente) équivaut donc à la feature parente d'un groupe de sous-features alternatives tandis que ces dernières représentent les variants de ce point. Il faut aussi noter qu'une feature optionnelle pourrait aussi correspondre à un point de variation spécifique. La feature en question correspondrait à la fois à ce point et à l'unique variant de ce point. Dans ce cas, le décideur pourrait avoir le choix de sélectionner ou de ne pas sélectionner ce variant, ce qui concorde bien avec la sémantique d'une feature optionnelle.

De plus, FODA introduit aussi deux mécanismes additionnels. Le premier, les règles de composition, permet de contraindre les combinaisons entre les différentes features du diagramme. Il existe deux types règles : "requiert" (pour être fonctionnelle, une feature pourra nécessiter une autre feature) et "exclut" (une feature ne pourra pas être fonctionnelle si une certaine feature est aussi sélectionnée). Le second mécanisme, les "*rationales*", permet de fournir des informations sur les relations entre les features. Ces "rationales" ont pour but de guider les acteurs (clients, ingénieurs software, ...) dans le processus de sélection des features.

Dans un diagramme, une feature représente donc une caractéristique, une fonctionnalité, ... d'un ou plusieurs des logiciels de la SPL. En sélectionnant une combinaison des ces features, un acteur construit une configuration correspondant à un des logiciels de la SPL. Cependant, cette sélection doit se faire en respectant les différentes contraintes (structure, features alternatives, règles de composition, ...) du diagramme. Lors de l'application engineering (voir Sous-section 2.4.2), en utilisant un diagramme de features, les acteurs (clients, développeurs, ...) sélectionnent donc les features nécessaires au produit souhaité. Le fait de sélectionner des features permet de fixer la variabilité (voir Sous-section 3.1.2).

Le diagramme de features présenté dans la Figure 3.5 permet d'illustrer les concepts décrits ci-dessus. Pour chaque nouvelle extension, l'exemple qui y est décrit sera raffiné et complété en fonction de l'amélioration proposée. Dans le cadre des SPLs, un diagramme de features est utilisé pour modéliser les logiciels d'une ligne de produits. Cependant, ce type de diagramme peut aussi servir à modéliser d'autres concepts : une voiture, un système d'alarme, ... Dans l'exemple, afin de faciliter la compréhension et de proposer un diagramme plausible, l'auteur de ce mémoire a décidé de quitter le domaine des logiciels pour proposer un modèle basé sur la configuration d'un ordinateur.

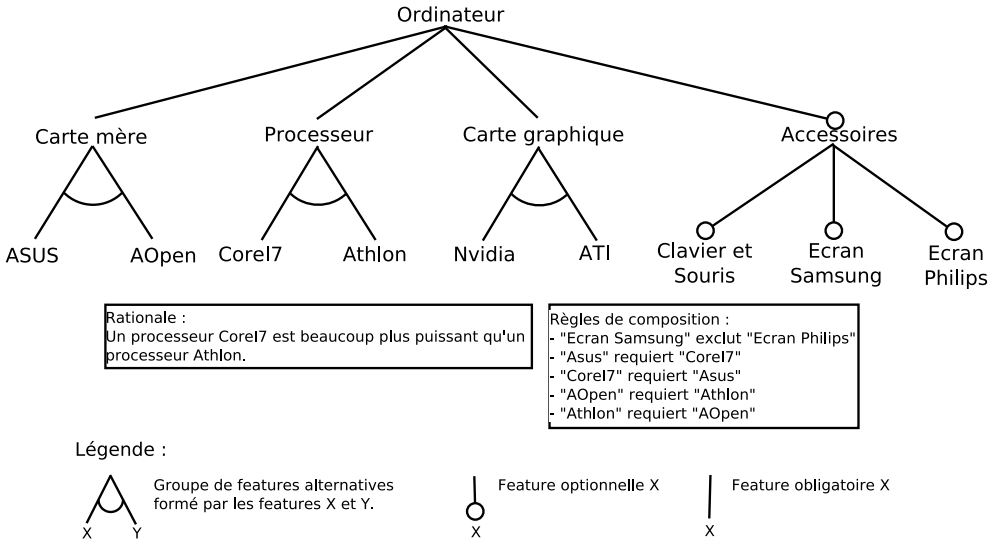

FIGURE 3.5 – Un diagramme de features représentant une gamme d'ordinateurs (syntaxe utilisée : [16]).

Dans ce diagramme, un ordinateur est obligatoirement composé de trois éléments : une carte mère, un processeur et une carte graphique (les composants non cités sont identiques pour chaque ordinateur et compris par défaut dans chaque configuration). De plus, un ordinateur peut aussi être accompagné d'accessoires (feature optionnelle). La feature "Carte mère" possède deux sous-features : "Asus" et "AOpen". Ces deux sous-features sont situées dans le même groupe de features alternatives. Ceci exprime le fait que ces sous-features sont une spécialisation de leur feature parente, "Carte mère". Il ne sera donc pas possible de sélectionner simultanément ces deux sous-features, le décideur (client,

commerçant, ...) devra donc en choisir une des deux. Les features "Processeur" et "Carte graphique" étant modélisées de la même manière que la feature "Carte mère", nous passerons directement à la feature "Accessoires". Cette dernière possède trois sous-features optionnelles. Les accessoires pouvant accompagner un ordinateur sont donc : un écran Samsung, un écran Philips et/ou un couple clavier/souris. Le décideur peut donc choisir d'acheter une certaine combinaison de ces accessoires. Cependant, toutes les combinaisons ne sont pas possibles, une règle de composition stipule que la feature "Ecran Samsung" exclut la feature "Ecran Philips" (il est sous-entendu que cette règle est aussi valable dans l'autre sens). Il ne sera donc pas possible de sélectionner simultanément ces deux sous-features. De manière similaire, les quatre autres règles ("Asus" requiert "CoreI7", "CoreIT" requiert "Asus", "AOpen requiert "Athlon" et "Athlon requiert "AOpen") imposent la sélection d'un certain type de processeur en fonction du choix de la carte mère et vice-versa. On peut donc imaginer que la carte mère Asus est exclusivement compatible avec le processeur CoreI7 tandis que la carte mère AOpen est elle uniquement compatible avec le processeur Athlon. Enfin, le "rationale" permet de guider le décideur dans sa sélection du processeur en indiquant que le processeur CoreI7 est le plus puissant.

Modélisation explicite des attributs

En 2002, Czarneski et al. publient un article relatant leur expérience de l'application de la programmation générative au développement de logiciels embarqués [12]. Dans le domaine des SPLs, le but ultime de ce type de programmation est de créer un générateur. A partir d'une liste de spécifications, ce type de logiciel est censé assembler différents artéfacts pour finalement produire un programme respectant ces spécifications. Ce qui est surtout intéressant ici, c'est que dans cet article, les auteurs apportent aussi une extension à FODA. Selon ces derniers, la sémantique de FODA ne leur a pas toujours permis d'exprimer tout ce qu'ils voulaient, ils ont donc décidé d'y apporter des améliorations. Celle qui nous intéresse, est celle permettant de modéliser explicitement les attributs des features.

Ce n'est pas nouveau, les features ont toujours pu posséder des attributs. Cependant, il n'existait pas de syntaxe permettant de les modéliser explicitement. Un attribut était donc exprimé comme une sous-feature obligatoire de la feature à laquelle il se rapportait. Cette modélisation n'était pas efficace. Dans le cas où des features possédaient plusieurs attributs, le diagramme pouvait vite être surchargé. De plus, il n'était pas non plus toujours facile de faire la différence entre les sous-features représentant réellement des features et les sous-features représentant des attributs. Dans [12], la notation utilisée pour modéliser les attributs est proche de celle des diagrammes de classe en UML. Chaque attribut peut avoir un type : entier, réel, booléen, ... Le diagramme illustré dans la Figure 3.6 (située sur la page suivante) est le même que celui de la Figure 3.5 excepté que certaines features disposent maintenant d'attributs.

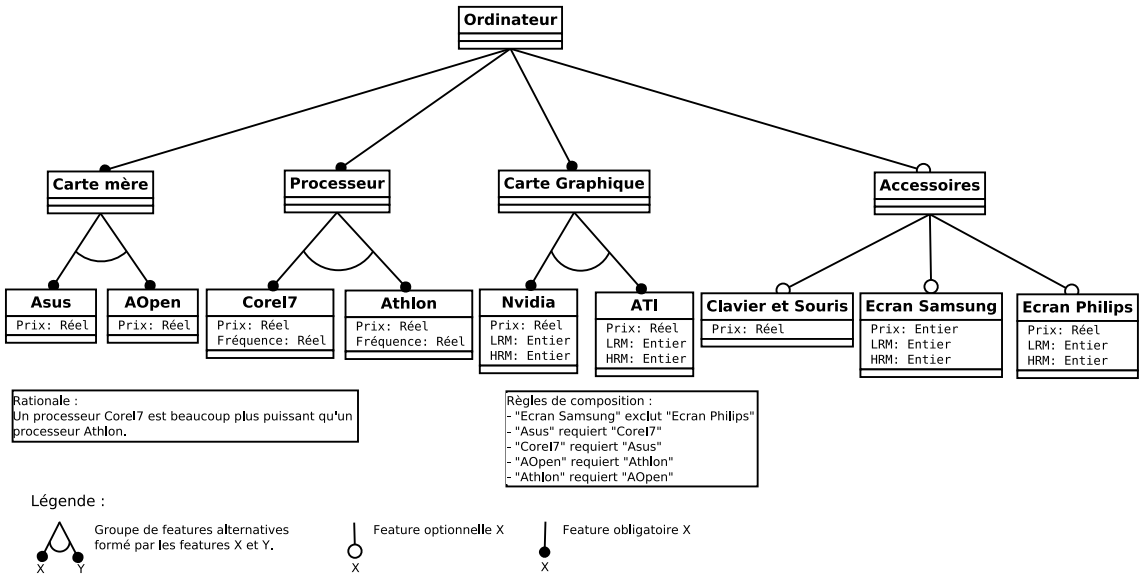

FIGURE 3.6 – Modélisation explicite d'attributs dans un diagramme de features (syntaxe utilisée : [12]).

Dans cette nouvelle version du diagramme, chaque sous-feature possède un attribut de type réel représentant son prix. De plus, les sous-features "CoreI7" et "Athlon" disposent chacune d'un second attribut spécifiant leur fréquence. Parallèlement, les sous-features "Nvidia", "ATI", "Ecran Samsung" et "Ecran Philips" détiennent deux attributs supplémentaires : LRM et HRM. Ils représentent la largeur (LRM) et la hauteur (HRM) maximales de résolution que la carte graphique / l'écran est capable de supporter. Grâce à cette nouvelle notation, les acteurs peuvent directement comprendre quels sont les attributs des différentes features. Cependant, comme nous pouvons le constater, malgré cette syntaxe spécifique, la modélisation des attributs diminue quand même la lisibilité. Ici, ce qu'il faut surtout retenir, c'est que pour la première fois, des chercheurs ont "officialisé" les attributs de features en permettant de les représenter explicitement.

Cardinalités et graphe acyclique orienté

Toujours en 2002, Riebisch et al. présentent un autre papier [23] proposant d'étendre FODA avec de nouvelles notations. Leur principale motivation vient du fait que la syntaxe utilisée pour modéliser les contraintes portant sur la sélection des sous-features d'un groupe est trop limitée et qu'elle peut parfois mener à certaines ambiguïtés. Par exemple, dans le diagramme (1) de la Figure 3.7 (située sur la page suivante), le choix peut se faire en deux fois et conduire à un résultat inattendu. Parmi le groupe de sous-features OR [2], un acteur peut par exemple sélectionner la sous-feature optionnelle "C". A cause du statut de

2. Equivalent de l'opérateur logique "OR", introduit par Czarnercki dans [11]. Dans un groupe de sous-features OR, il est donc obligatoire de sélectionner au minimum une sous-feature.

cette dernière, il pourra ensuite choisir de ne pas la sélectionner ce qui reviendra à n'avoir choisi aucune sous-feature du groupe. Ceci est donc contraire au mécanisme du groupe de features OR qui requiert la sélection d'au moins une feature. Pourtant, la notation utilisée est tout à fait correcte. Afin de solutionner ce problème, il est nécessaire de remodéliser le diagramme (1) en (2), ce qui permet de rendre toutes les contraintes explicites (dans cette nouvelle version du diagrammes, l'acteur comprend clairement qu'il peut ne sélectionner aucune sous-feature de la feature parente "X"). Ce processus se nomme la "normalisation".

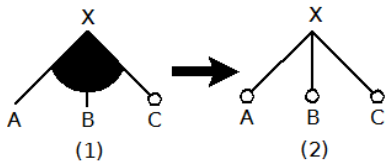

FIGURE 3.7 – Modélisation ambigüe d'un groupe de features OR.

L'amélioration introduite permet de modéliser les contraintes portant sur la sélection des sous-features d'un groupe de façon complète et non ambigüe. Cette extension est directement inspirée du système des multiplicités utilisé dans les diagrammes de classe en UML. Pour chaque groupe de sous-features, elle permet d'indiquer le nombre minimum et le nombre maximum de sous-features qui peuvent être sélectionnées. Cependant, il faut tout de même préciser que ce mécanisme n'est pas nouveau dans les diagrammes de features, il était simplement utilisé de manière implicite et limité. Par exemple :

- Un groupe de sous-features alternatives possède une multiplicité 1..1, il est obligatoire de sélectionner au minimum et au maximum une sous-feature.
- Un groupe de X sous-features optionnelles possède une multiplicité 0..X, n'importe quelle combinaison de ces sous-features peut être sélectionnée.
- Un groupe de X sous-features obligatoires possède une multiplicité X..X, toutes les sous-features doivent être sélectionnées.
- Un groupe OR de X sous-features possède une multiplicité 1..X. Les acteurs peuvent sélectionner au minimum une sous-feature et au maximum X sous-features.
- ...

La nouvelle notation va donc venir enrichir la sémantique des diagrammes de features en permettant la modélisation explicite de nouvelles multiplicités : 0..n, 1..n, n..m, m..* où $1 \leq n \leq m \leq *$ (avec $*$, le nombre de sous-features composant un groupe). Grâce à ces nouvelles multiplicités, il va être possible de représenter des situations plus complexes nécessitant des multiplicités spécifiques autres que celles qu'il était déjà possible de modéliser implicitement.

La modélisation des multiplicités n'est pas la seule amélioration présentée dans ce papier, les auteurs proposent aussi de modéliser les features non plus dans un arbre, mais dans un graphe acyclique orienté. Ceci permet donc à une sous-feature de posséder plusieurs features parentes. Le fait que le graphe soit acyclique permet de s'assurer qu'il n'existe pas de cycle entre une sous-feature et ses différents ancêtres, une sous-feature ne peut donc pas se retrouver parente d'une de ses features ancêtres. Cette extension peut par exemple servir à

simplifier un diagramme en réduisant à un exemplaire le nombre de fois où une sous-feature commune à plusieurs features est représentée. Dans le diagramme (1) de la Figure 3.8, malgré le fait que la sous-feature E soit identique pour les features A, B et C, elle a dû être représentée trois fois à cause de la structure en arbre. A l'inverse, dans le diagramme (2), grâce au graphe, la sous-feature E n'a dû être représentée qu'une seule fois.

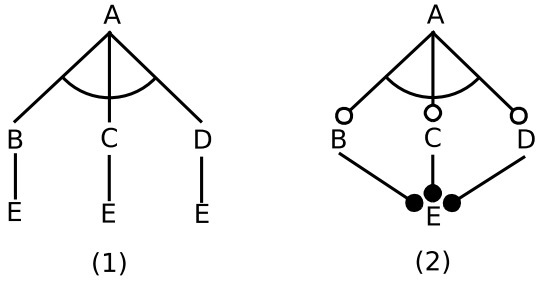

FIGURE 3.8 – Les avantages d'un graphe orienté acyclique par rapport à un arbre.

Tout comme précédemment, le diagramme de la Figure 3.5 a été repris pour y inclure les extensions présentées dans [23]. Le résultat est visible dans la Figure 3.9.

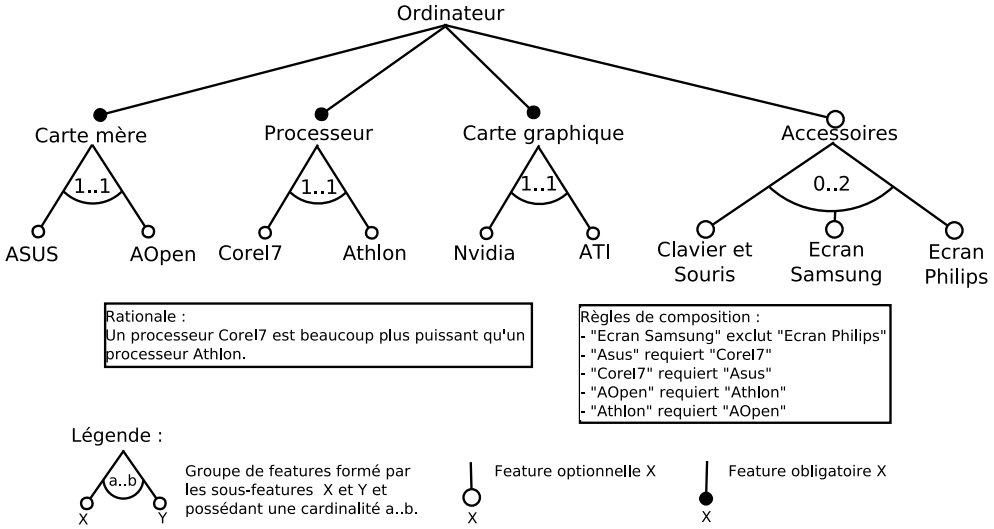

FIGURE 3.9 – Un diagramme de features sous forme de graphe acyclique orienté utilisant le mécanisme des multiplicités (syntaxe utilisée : [23]).

Dans ce diagramme, les groupes de sous-features des quatre principales features (”Carte mère”, ”Processeur”, ”Carte graphique” et ”Accessoires”) possèdent maintenant des multiplicités. La multiplicité 1..1 des trois premiers groupes contraint à ne sélectionner qu'une et une seule sous-feature de chaque groupe. De manière similaire, la cardinalité 0..2 du dernier groupe limite la sélection à maximum deux sous-features. Cette cardinalité n'était pas vraiment nécessaire,

le statut optionnel de chacune des sous-features et la première règle de composition garantissant déjà que maximum deux sous-features puissent être choisies. Enfin, le diagramme adopte la structure d'un graphe acyclique orienté et non plus celle d'un arbre. Etant donné qu'aucune sous-feature ne possède plusieurs features parentes, cette nouvelle structure n'est pas apparente sur le diagramme.

Il faut aussi noter que la syntaxe utilisée ici [23] est quelque peu différente de celle employée dans l'extension [12] présentée dans la sous-section précédente. En effet, ici, les sous-features des groupes sont considérées comme optionnelles. A partir du moment où un décideur a sélectionné une telle feature, il est obligé de la prendre, il ne peut plus jouer sur son statut optionnel (le choix se fait en une seule étape et non plus en deux. Voir Figure 3.7). Dans l'extension précédente, ces sous-features étaient considérées comme obligatoires mais les contraintes de sélection de groupe (features alternatives ou features OR) empêchaient de toutes les sélectionner. Ces différences syntaxiques démontrent que dans les différentes extensions de FODA, la syntaxe est quelquefois réutilisée sans vraiment faire attention à la sémantique qui l'accompagne. Ceci est une des lacunes majeures des diagrammes de features abordée dans le chapitre suivant.

Enfin, il est aussi bon de préciser qu'il y a eu une évolution de vocabulaire à partir de [13], les scientifiques ont commencé à utiliser le terme cardinalité comme synonyme du terme multiplicité.

Formulation des contraintes à l'aide d'un langage formel (2003)

En 2003, Streitferdt et al. [26] proposent [3] d'exprimer certaines relations grâce au langage de contraintes OCL [4] ("Object Constraint Language"). Auparavant, les relations entre les features pouvaient être exprimées de trois manières différentes. Premièrement, grâce à la structure en arbre ou en graphe. Deuxièmement, grâce à l'utilisation des cardinalités de groupe. Et enfin, grâce aux contraintes "requiert" et "exclut". Selon les auteurs, ces mécanismes étaient incomplets et ne permettaient pas d'exprimer toutes les relations possibles, surtout depuis que les attributs et leurs types avaient été introduits [26]. De plus, de leur côté, certains scientifiques définissaient de nouvelles relations sans toujours clairement définir la sémantique de ces dernières. Il était donc souvent compliqué de vérifier automatiquement (à l'aide d'un logiciel) la cohérence (si l'ordinateur ne connait pas la sémantique de la relation définie par le scientifique, il ne pourra pas vérifier sa cohérence) de toutes les contraintes d'un diagramme de features. Ils ont donc décidé d'exprimer les relations grâce au très connu langage de contraintes : OCL. Ce dernier est utilisé dans les diagrammes UML pour exprimer des contraintes qu'il n'est pas possible de représenter avec des éléments graphiques.

Dans ce langage, une contrainte est composée de quatre sections. La première, "`context`", contient les éléments du graphique qui seront affectés par la contrainte. La seconde, "`inv`", est composée d'un invariant qui devra toujours être vérifié. Dans le cas contraire, la contrainte sera considérée comme violée. Enfin, les deux dernières sections, "`pre`" et "`post`", représentent respectivement

3. Ces auteurs ne sont pas les premiers à proposer d'exprimer les contraintes à l'aide d'OCL (par exemple, les rédacteurs de [23] suggéraient aussi d'utiliser ce langage). Cependant, ils sont les premiers à publier un papier consacré à ce problème et proposant une version d'OCL adaptée aux diagrammes de features.

4. http ://www.omg.org/spec/OCL/

des pré-conditions et des post-conditions pour les méthodes de classes (dans les diagrammes de classes). Le langage utilisé pour construire les expressions des différentes sections est défini à l'aide d'une grammaire élémentaire [5].

Afin de pouvoir utiliser OCL pour compléter les diagrammes de features, les auteurs ont dû apporter quelques modifications au langage de contraintes. Ils ont tout d'abord écarté les deux dernières sections, "`pre`" et "`post`", car elles n'étaient pas nécessaires. Parallèlement, ils ont rajouté deux nouveaux constructeurs :

1. `selected(<feature>)` : Indique, si oui ou non, la feature "`feature`" du diagramme a été sélectionnée.
2. `value(<feature>, <type>)` : Retourne la valeur de la feature "`feature`" respectant le type "`type`". Il faut savoir que dans ce papier, les auteurs ont utilisé une version des diagrammes de features où les attributs des features sont modélisés comme des sous-features spécifiques : les paramètres. Chacun de ces paramètres possède un type et peut être instancié à une valeur respectant ce type.

Cette version modifiée d'OCL a été baptisée A-OCL. A partir de là, il a donc été possible de spécifier toutes les contraintes dites additionnelles, c'est-à-dire, celles que l'on ne peut pas exprimer en utilisant la structure du diagramme ou les cardinalités de groupe. "Requiert" et "exclut" sont des exemples de telles contraintes, il a d'ailleurs été possible de les redéfinir en utilisant A-OCL. Par exemple, le code A-OCL de la Figure 3.10 possède la même sémantique que l'instruction "exclut" :

```
context F1, F2
inv requires :
    (selected(F1) implies not selected(F2))
    and
    (selected(F2) implies not selected(F1))
```

FIGURE 3.10 – Code A-OCL équivalant à la contrainte "exclut".

Dans la section contexte, on a donc précisé les deux features affectées par la contrainte : "F1" et "F2". Dans l'invariant, on stipule que la sélection de "F1" implique la non-sélection de "F2" et vice-versa. Grâce à A-OCL, il est donc maintenant possible de définir de nouvelles contraintes, faisant non seulement intervenir des features mais aussi des attributs de features. De plus, pour les attributs, il est aussi possible de spécifier des assignations de valeur sous forme de contraintes. Le code A-OCL qui suit la Figure 3.11 (située sur la page suivante) illustre ce principe.

Le sous-diagramme de cette figure a été extrait du diagramme principal de la Figure 3.6. La différence est que la feature "Carte_mère" possède maintenant aussi un attribut réel correspondant à son prix. Ce prix sera bien sûr égal au prix de la sous-feature ("Asus" ou "AOpen") sélectionnée. Les trois contraintes qui suivent la figure permettent de le calculer automatiquement.

5. Pour retrouver la spécification complète d'OCL, il suffit de se rendre sur le site : `http://www.omg.org/technology/documents/formal/ocl.htm`

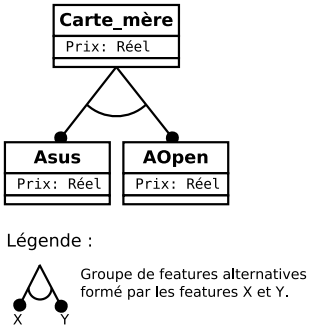

FIGURE 3.11 – Sous-diagramme extrait du diagramme de la Figure 3.5 (syntaxe utilisée : [26]).

```
context Carte_mère, Asus, AOpen
inv prix_Carte_mère :
    Carte_mère:Prix = Asus:Prix + AOpen:Prix

context Asus
inv prix_Asus :
    if selected(Asus) then
       Asus:Prix = 350
    else
       Asus:Prix = 0

context AOpen
inv prix_AOpen :
    if selected(AOpen) then
       AOpen:Prix = 250
    else
       AOpen:Prix = 0
```

La première contrainte exprime le fait que le prix de "Carte_mère" est égal à la somme du prix de ses deux sous-features. A première vue, ce calcul peut sembler erroné. En effet, sur le sous-diagramme de la Figure 3.11, la contrainte du groupe de features alternatives impose de ne sélectionner qu'une seule sous-feature. Pourtant, le calcul impose de payer le prix conjugué des deux sous-features. Pour comprendre, il faut aussi prendre en compte les deux dernières contraintes. La première exprime le fait que le prix de la feature "Asus" est égal à 350 si elle est sélectionnée et à 0 dans le cas contraire. La deuxième, celle concernant "AOPen", est aussi basée sur le même mécanisme. Dans tous les cas, le prix de "Carte_mère" sera donc bien égal au prix de la sous-feature sélectionnée. Grâce à ce mécanisme, il est aussi possible de calculer le prix total d'un ordinateur. L'ensemble des contraintes A-OCL nécessaires à ce calcul peut être retrouvé dans l'annexe A.

L'ajout d'un langage permettant de modéliser les contraintes additionnelles permet de dépasser les limites graphiques des diagrammes de features. En effet, il n'est plus nécessaire de se limiter aux contraintes classiques "exclut" et "requiert". En fonction de leurs besoins, les ingénieurs domaine peuvent main-

tenant définir de nouvelles contraintes. De plus, A-OCL permet de clairement modéliser des relations comprenant des attributs et leurs valeurs. Enfin, il ne faut pas oublier que A-OCL est un langage formel (possédant une syntaxe et une sémantique clairement définies), il est donc compréhensible par un ordinateur, ce qui permet à tout moment de vérifier la cohérence des contraintes. Si les contraintes avaient été modélisées à l'aide d'un langage naturel (le français par exemple), elles auraient sans doute été plus faciles à comprendre par un humain mais impossible à vérifier par un logiciel.

Ajout d'un mécanisme d'inclusion

Dans certaines entreprises, le nombre de features inclues dans un diagramme peut être très élevé (il peut y avoir plusieurs centaines de features). Représenter toutes ces features dans un seul et même diagramme se révèle très compliqué voire même impossible. Les ingénieurs domaine devraient élaborer un diagramme qui serait complet (e.g. reprenant toutes les features) tout en étant lisible et compréhensible. Cependant, dans [27] [6], Bednasch propose un mécanisme permettant de simplifier les gros diagrammes de features : la modularisation. Ce dernier permet, grâce à une feature spécifique, d'inclure dans un diagramme de features, un autre diagramme externe au premier. L'exemple de la Figure 3.12 (située sur la page suivante) permet d'illustrer cette nouvelle extension. Dans la syntaxe utilisée, les features obligatoires possèdent maintenant un signe distinctif clair : un cercle rempli au-dessus de leur nom.

Dans ce diagramme, la feature "Accessoires", ses sous-features et la règle de composition les concernant ont été externalisées, elles représentent maintenant un diagramme de features à part entière. Ce dernier est inclus dans le diagramme de features principal grâce à la feature spécifique "Accessoires". L'étoile à côté du nom de cette feature indique qu'elle représente un diagramme de features externe. La modularisation permet donc de simplifier les diagrammes tout en augmentant la lisibilité grâce à l'externalisation de features sous forme de nouveaux diagrammes.

6. Cet article étant en allemand, l'auteur de ce mémoire s'est surtout basé sur un autre article en anglais [13] qui en parlait.

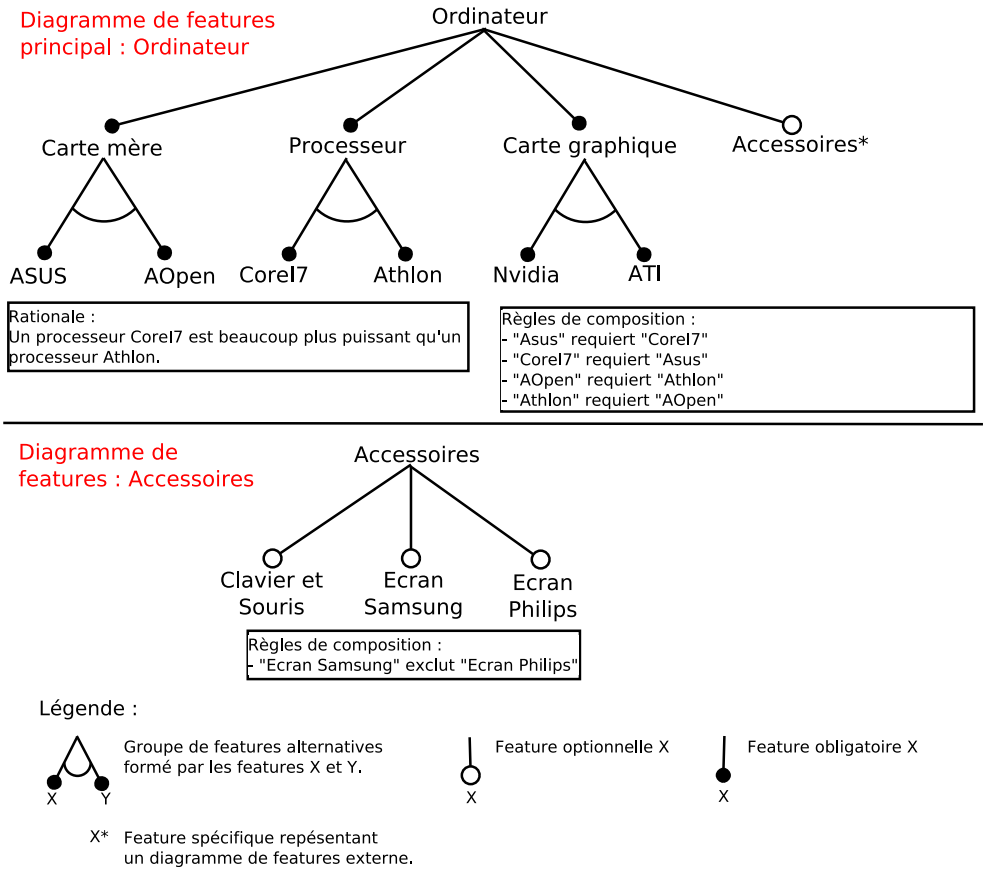

FIGURE 3.12 – Exemple d'utilisation d'un mécanisme d'inclusion (syntaxe utilisée : [27]).

Récapitulatif des extensions présentées

En résumé, les extensions présentées ci-dessus ont donc permis d'améliorer FODA grâce :

- A la modélisation explicite des attributs : avant, les attributs étaient modélisés sous forme de features. Maintenant, il devient possible de les modéliser explicitement grâce à une syntaxe spécifique.
- A la modélisation des cardinalités : cette extension permet d'exprimer toutes sortes de cardinalités de groupe, ne limitant plus les possibilités aux traditionnelles contraintes de groupe (groupes de features alternatives et groupes OR).
- A la possibilité d'exprimer les diagrammes de features non seulement sous forme d'arbres mais aussi sous forme de graphes acycliques orientés : ce type de structure peut par exemple simplifier un diagramme ou permettre à une sous-feature de posséder simultanément deux features parentes.
- A la formulation des contraintes portant sur des features et/ou des attributs à l'aide d'un langage formel : au départ, FODA ne permettait l'expression que de contraintes basiques, "requiert" et "exclut". Grâce à cette

nouvelle extension, il est maintenant possible d'exprimer toutes sortes de contraintes, que ce soit sur des features ou des attributs.
- A l'introduction d'un mécanisme d'inclusion : dans le cas où un diagramme contient un nombre élevé de features, il devient par exemple possible de le subdiviser en plusieurs sous-diagrammes. Dans le diagramme principal, il ne suffira alors plus que d'inclure, grâce à des features spécifiques, chacun de ces sous-diagrammes.

Quelques pistes pour la découverte des autres extensions de FODA

Comme précisé plus haut, seules les extensions ayant été réutilisées dans la technique de modélisation du chapitre suivant ont été présentées. Cependant, depuis FODA, d'autres extensions ont été introduites. Ces dernières essaient de solutionner des problèmes ou de combler des vides rencontrés dans les diagrammes de features :

- Dans [17], les auteurs introduisent FORM, la première extension de FODA. Cette dernière permet de classifier les features à l'aide d'un système de quatre couches/niveaux d'abstraction. De plus, elle introduit aussi trois nouveaux types de relations entre features.
- Dans [30], les auteurs présentent une extension permettant de modéliser des features externes (fonctionnalités ou possibilités offertes par le système où est déployé le produit) et de spécifier les binding times (voir Sous-section 3.1.5) des points de variation.
- Dans [12], les auteurs introduisent les cardinalités de features. Cette extension permet de sélectionner plusieurs fois une même feature. Chaque sélection correspond à une variation de la feature (où certaines sous-features ont été selectionnées et d'autres non, où les attributs possèdent des valeurs particulières, ...). La cardinalité de la feature permet de limiter le nombre de sélections de cette dernière en fournissant une borne minimum et maximum.
- Dans la méthode RequiLine [32], les diagrammes de features peuvent utiliser de nouveaux types de relations et les features peuvent posséder des attributs.

Ces extensions ne sont que des exemples parmi d'autres. Les papiers [3], [7], [24] et [28] constituent de bonnes pistes pour en apprendre plus sur les extensions apportées à FODA.

3.2.2 Autres techniques de modélisation de la variabilité

Les diagrammes de features sont un moyen spécifique de modéliser la variabilité. Parallèlement, il existe d'autres techniques de modélisation. Les différents auteurs classifient souvent ces dernières en fonction des concepts ou des éléments sur lesquels elles se basent. L'objectif de ce mémoire ne portant pas sur ces techniques, l'auteur ne fera que citer certaines des catégories de ces techniques accompagnées de un ou deux exemples. Pour plus d'information, les papiers [3], [7] et [28] donnent un aperçu et une appréciation de certaines de ces techniques.

Techniques axées sur l'architecture [3]

Dans cette catégorie, on peut citer Koala [31]. Ce dernier est à la fois un modèle de composants et un langage de description d'architectures. La partie langage inclut différents mécanismes permettant de modéliser la variabilité. Par exemple, chaque logiciel est caractérisé par un ensemble de paramètres et ces derniers peuvent servir à déterminer les fonctionnalités du système. La variabilité peut donc être modélisée grâce à ces paramètres. En fonction de ses paramètres, un programme possédera certaines fonctionnalités. On peut aussi mentionner xADL 2.0 [14], ce framework permet de créer des langages de description d'architectures basées sur XML. Certains éléments de modélisation offerts par xADL 2.0 permettent de modéliser la variabilité. Par exemple, les types de variant ("variant types") permettent de représenter un choix entre minimum deux éléments.

Techniques basées sur les Use Case [28]

A la base, les use cases ne possèdent pas tous les mécanismes permettant de modéliser "explicitement" et "efficacement" la variabilité. Des scientifiques ont donc publié des articles dans lesquels ils proposent d'étendre la notation des use cases. Dans [5], les auteurs étendent la notation textuelle des use cases à l'aide d'un système de tags. Chaque description textuelle d'un use case contient donc un ensemble de tags et ces derniers peuvent être vus comme des points de variation. Par exemple, dans un texte, chaque occurrence d'un tag devra être remplacée par un des "variants" de ce tag. Dans [15], les auteurs étendent la notation graphique des use cases. Chaque use case peut être dit "kernel" (utilisé par tous les produits de la SPL), optionnel ou encore variant. Ces derniers sont des use cases spécifiques possédant une version spécifique pour certains des produits de la SPL.

Basées sur les diagrammes de classes [28]

Ici aussi, les scientifiques ont décidé d'étendre la notation des diagrammes de classes afin d'y inclure des mécanismes permettant de modéliser la variabilité. L'utilisation des diagrammes de classes peut se révéler intéressante pour modéliser la variabilité dans les données. Par exemple, dans [9], afin de modéliser les points de variation, les auteurs utilisent le stéréotype "variation point" tandis que pour les variants, ils utilisent le stéréotype "variant". La relation entre un point de variation et ses variants est représentée grâce à la relation d'héritage des diagrammes de classes.

Chapitre 4

TVL : "*a text-based variability language*"

Dans le chapitre précédent, nous nous sommes intéressés à la variabilité et au moyen de la modéliser à l'aide des diagrammes de features. Dans le présent chapitre, nous allons dresser les lacunes de ces diagrammes et présenter TVL, un langage développé par l'équipe LIEL des FUNDP afin de combler ces lacunes et de fournir au monde industriel une solution en adéquation avec ses besoins. TVL est un langage de modélisation basé sur les features. Contrairement aux diagrammes, il n'utilise aucune notation graphique, les features, les attributs et les contraintes sont spécifiés textuellement (d'où le nom de TVL : "a text-based variability language"). Dans le contexte de TVL, il ne faudra plus parler de diagrammes de features mais de modèles de features.

La première section de ce chapitre décrira les principaux défauts des diagrammes de features. La deuxième section sera subdivisée en deux sous-sections. La première sera un rappel théorique concernant les grammaires formelles tandis que la seconde présentera les fonctionnalités et la grammaire de TVL. Enfin, la dernière section décrira les différents avantages de TVL.

4.1 Inconvénients des langages de features actuelles

Malgré le fait que la représentation graphique des features présente des avantages [19] : compréhensible par la très grande majorité des acteurs, concepts facilement "manipulables", ... Dans le domaine industriel où les SPLs peuvent être composées de centaines de features, elle possède toutefois certaines lacunes. Ci-dessous, les trois défauts majeurs des diagrammes de features sont explicités.

Les limites graphiques

Dans le domaine industriel, les SPLs peuvent inclure des centaines de features. Dès lors, l'utilisation des diagrammes peut ne pas être indiquée [6] [19]. Vu l'impossibilité de représenter l'ensemble des features en un seul schéma, les ingénieurs vont souvent être obligés de créer plusieurs sous-diagrammes. De plus, l'ajout d'éléments tels que les attributs ou les contraintes graphiques va

avoir comme conséquence d'alourdir encore un peu plus les diagrammes. Dans certaines situations, un ingénieur pourra se retrouver face à des centaines de features subdivisées en plusieurs sous-diagrammes surchargés par différents éléments. La compréhension, la gestion et l'exploration de ce genre de diagrammes pourront vite devenir très complexes. Par exemple, si une contrainte fait intervenir des features et leurs attributs répartis sur plusieurs sous-diagrammes, l'ingénieur va être obligé de jongler entre plusieurs documents. Si c'est le cas pour plusieurs contraintes, il pourra être amené à passer plus de temps à trouver les documents qu'à gérer ou comprendre le diagramme. Dans ces conditions, garder un schéma mental clair du diagramme est très difficile.

Pour les ingénieurs, l'aspect graphique peut ne représenter un avantage que durant l'analyse des exigences du client. Avant, ils pourront par exemple préférer encoder les diagrammes à l'aide d'un langage formel. Un tel langage leur paraîtra sans doute plus intuitif tout en permettant de vérifier automatiquement (à l'aide d'un logiciel) certains éléments ou qualités (par exemple la cohérence) du diagramme. Dans [17], Kang et al. expliquent que leurs diagrammes de features peuvent vite devenir complexes. Dès lors, ils préfèrent encoder un modèle de features textuellement et utiliser la représentation graphique pour analyser des sous-ensembles de features du modèle. Enfin, il ne faut pas oublier de préciser que ces problèmes graphiques ne sont pas seulement propres aux diagrammes de features, d'autres techniques de modélisation graphique rencontrent le même genre de difficultés [19].

La définition de la sémantique

Ce second problème ne provient pas des diagrammes de features mais des auteurs qui en définissent les langages de modélisation. En effet, dans bon nombre de cas, les scientifiques définissent la syntaxe graphique de leur langage tout en omettant d'en définir la sémantique [21]. Pour certains, la sémantique est considérée comme intuitive, il n'y aurait donc pas besoin de la spécifier. Or, la définition de la sémantique est primordiale. Elle a par exemple comme avantage de permettre la comparaison de l'expressivité de deux langages. Dans le cas de sémantiques identiques, elle permet de détecter les constructions syntaxiques les plus pratiques. De plus, sans sémantique, un langage ne peut pas être analysé automatiquement par un ordinateur. Malgré le fait que ce défaut soit dû aux auteurs et non aux diagrammes, étant donné qu'il se retrouve dans nombre de langages de diagrammes de features, il peut être considéré comme un problème récurrent des diagrammes de features.

Besoin d'outils dédiés

De par leur nature graphique, les diagrammes de features ne peuvent être manipulés qu'à l'aide d'outils dédiés [6] [19]. Il sera sans doute toujours possible de créer un diagramme de features en utilisant un logiciel de dessin classique mais quand il s'agira de manipuler ce diagramme, les choses se corseront, surtout s'il est composé de nombreuses features. Par exemple, pour déplacer ou inclure une feature, il faudra souvent redessiner "manuellement" tout le diagramme. De leur côté, les outils dédiés sont donc prévus pour gérer les diagrammes de features, le déplacement ou l'inclusion d'une feature ne posera aucun problème, le graphe pourra par exemple être restructuré automatiquement. Ici, le problème,

c'est qu'en utilisant les diagrammes de features, les utilisateurs deviennent dépendants de la technologie. Sans ces outils, ils ne pourront pas exploiter efficacement les diagrammes.

4.2 Grammaire et fonctionnalités-clés de TVL

TVL est un langage qui a été développé afin de combler certaines des faiblesses (présentées dans la section précédente) des diagrammes de features. Grâce à sa syntaxe et sa sémantique, il permet d'exprimer simplement des modèles de features très complets. Dans cette section, la présentation des fonctionnalités et de la grammaire de TVL sera d'abord précédée d'un bref rappel théorique concernant les grammaires formelles.

4.2.1 Rappel théorique : les grammaires formelles

Tout comme le français ou d'autres langages naturels, les langages formels (les langages de programmation, ...) possèdent chacun une grammaire. Une telle grammaire est constituée d'un ensemble de règles, appelées règles de production ou de réécriture, précisant comment les mots du langage peuvent être construits et structurés. Ces règles sont elles-mêmes formées de symboles terminaux et non-terminaux. Les symboles terminaux sont souvent des caractères ou des suites de caractères. Par exemple : "`while`", "`+`", "`if`", ... Quant aux symboles non-terminaux, ils correspondent chacun à une règle de production. Pour faciliter la compréhension, la coutume veut que les terminaux soient écrits en minuscules et les non-terminaux en majuscules. De plus, afin de pouvoir différencier les terminaux des constructeurs syntaxiques (les égalités des règles de production, ...) propres à la grammaire, ceux-ci sont encadrés de guillemets. Ci-dessous, voici une grammaire formée de deux règles de production :

```
(a) EXPRESSION =  EXPRESSION "+" EXPRESSION
                | NATUREL

(b) NATUREL =  "0"
             | ["1"-"9"]["0"-"9"]*
```

Le nom du non-terminal auquel correspond une règle se situe à gauche de l'égalité. A droite se trouve le corps de la règle, il décrit comment le non-terminal peut être réécrit (l'action de remplacer un non-terminal par la partie droite de la règle lui correspondant est appelée réécriture). Le caractère "`|`" signifie que la suite de symboles se trouvant derrière lui représente une réécriture alternative. Le corps d'une règle peut être composé de plusieurs réécritures alternatives (chacune précédée du symbole "`|`"). Dans cette grammaire, la règle (a) stipule que le non-terminal "`EXPRESSION`" peut être réécrit de deux façons différentes : soit en une somme de deux "`EXPRESSION`" (un non-terminal peut donc lui-même être inclus dans le corps de sa propre règle), soit en un autre non-terminal : "`NATUREL`". La seconde règle (b) correspond au non-terminal "`NATUREL`". Elle précise qu'un tel symbole peut être réécrit de deux manières différentes. Premièrement, "`NATUREL`" peut tout simplement être réécrit en "`0`". Deuxièmement, il peut aussi être réécrit en un nombre dont le premier caractère serait situé dans un intervalle entre un et neuf et dont les caractères suivants seraient n'importe

quelle combinaison de chiffres compris entre zéro et neuf. Le nombre de chiffres de cette combinaison peut très bien être égal à zéro (dans ce cas, la combinaison est vide et n'existe tout simplement pas). En effet, le caractère "`*`" signifie que le symbole auquel il se raccroche peut ne pas être répété, ou au contraire, répété un nombre quelconque de fois. Le caractère "`+`", est semblable à "`*`" sauf que le symbole auquel il est associé doit être répété au moins une fois. Ainsi, par exemple, les nombres :

- `0`, `1560` et `9` sont des naturels, ils respectent la première ou la seconde altenative de la règle de production (b).
- `0156` et `09` ne sont pas des naturels, ils violent les deux alternatives de la règle de production (b).

Génération des mots

Grâce à ses règles de production, une grammaire est capable de générer des mots. Pour cela, il faut partir de l'axiome (le non-terminal à partir duquel tous les mots sont générés) et appliquer les règles de production jusqu'à obtenir une chaîne de symboles composée uniquement de symboles non-terminaux.

```
EXPRESSION

EXPRESSION + EXPRESSION

NATUREL + EXPRESSION

237 + EXPRESSION

237 + NATUREL

237 + 72
```

FIGURE 4.1 – Exemple de dérivation

Dans la Figure 4.1, le non-terminal "`EXPRESSION`" a d'abord été réécrit en utilisant la première alternative de la règle (a). Ensuite, durant la seconde étape, le non-terminal "`EXPRESSION`" le plus à gauche a été réécrit à l'aide de la seconde alternative de la règle (a). A la troisième étape, le nouveau non-terminal "`NATUREL`" a été réécrit grâce à la seconde alternative de la règle (b). Enfin, le non-terminal de droite "`EXPRESSION`" a lui aussi été réécrit en utilisant successivement les secondes alternatives des règles (a) et (b). L'application d'une suite de règles de réécriture afin de générer un mot est appelée dérivation. La dérivation de l'exemple ci-dessous est notée `EXPRESSION` $\Longrightarrow$ `237 + 72`. A partir de l'axiome "`EXPRESSION`", il a été possible de dériver le mot `237 + 72`. L'ensemble des mots qu'une grammaire G est capable de générer est appelé langage de G et est noté $L(G)$.

Reconnaissance des mots

Dans la Figure 4.1, le mot `237 + 72` a été généré à partir des règles de production. Cependant, il peut aussi être intéressant de savoir déterminer si oui ou non, une suite de caractères fait partie du langage défini par une grammaire et forme alors un mot. Les logiciels capables d'accomplir ce travail sont appelés reconnaisseurs ou analyseurs. Ces derniers divisent l'analyse d'une chaîne de caractères en deux phases [1].

La première, l'analyse lexicale, consiste à regrouper les caractères afin de former des tokens. Chacun d'eux correspond à un symbole terminal de la grammaire. La seconde, l'analyse syntaxique, va devoir déterminer si oui ou non, la suite de symboles transmise par l'analyse lexicale est dérivable à partir des règles de la grammaire. Autrement dit, il va falloir reconstruire, en partant de cette suite, la dérivation permettant de la générer. Dans le cas où il existe bien une dérivation, la chaîne analysée fait alors partie du langage de la grammaire. Dans le second cas, elle n'en fait tout simplement pas partie. Lorsqu'un mot est dérivable à partir de plus d'une seule seule dérivation (e.g. il existe plusieurs dérivations permettant de générer le mot), la grammaire est dite ambigüe.

Lors de la construction de la dérivation de la Figure 4.1, nous avons appliqué une stratégie LL ("Left to right, Leftmost derivation"). A chaque étape, nous avons parcouru la chaîne de gauche à droite en réécrivant systématiquement le non-terminal le plus à gauche, ce qui a produit une dérivation gauche. A l'inverse, dans une stratégie LR ("Left to right, Rightmost derivation"), la chaîne est toujours parcourue de gauche à droite mais c'est le non-terminal le plus à droite qui est toujours réécrit, ce qui produit une dérivation droite. De par la structure [2] de sa grammaire, un langage ne peut par exemple être reconnu que par un reconnaisseur LL (générant des dérivations gauches) ou LR (générant des dérivations droites). La grammaire d'un tel langage est alors dite grammaire LL ou LR.

Arbre syntaxique

Il faut aussi noter que toute dérivation peut être exprimée sous la forme d'un arbre syntaxique. Par exemple, la dérivation de la Figure 4.1 est représentée par l'arbre syntaxique de la Figure 4.2 (située sur la page suivante).

Dans ce type d'arbre, chaque noeud représente un symbole non-terminal tandis que chaque feuille incarne un symbole terminal. Chaque non-terminal est relié aux symboles d'une de ses alternatives de réécriture par des branches (en fonction de l'alternative qui a été choisie durant la dérivation). Dans cet arbre, le noeud racine est donc l'axiome de la grammaire : "`EXPRESSION`". Ce noeud possède trois fils correspondant aux symboles de la première alternative de la règle (a). Le fils "`EXPRESSION`" le plus à gauche possède lui-même son propre fils, il correspond au symbole "`NATUREL`" de la seconde alternative de la règle (a). Enfin, ce noeud "`NATUREL`" possède aussi un fils, "`237`", le symbole terminal de la seconde alternative de la règle (b). De manière similaire, le second fils "`EXPRESSION`" (le plus à droite) du noeud racine possède aussi un fils "`NATUREL`"

1. Ces deux phases sont expliquées plus en détails dans la Section 6.1 du chapitre suivant.
2. L'objectif de ce mémoire n'étant ni les grammaires, ni les reconnaisseurs, son auteur invite le lecteur à consulter le site `http://www.enseignement.polytechnique.fr/informatique/profs/Luc.Maranget/compil/poly/grammatical.html` pour obtenir plus de renseignements.

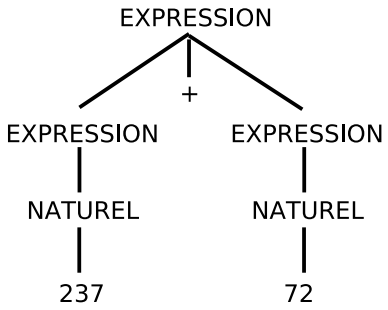

FIGURE 4.2 – Arbre syntaxique correspondant à la dérivation de la Figure 4.1.

qui, lui-même, a comme fils "72", le symbole terminal de la seconde alternative de la règle (b). Cet arbre syntaxique correspond donc bien à la dérivation de la Figure 4.1.

Définitions formelles

Après cette présentation "pratique" des grammaires, il est maintenant possible d'en donner une définition formelle.

Définition 4.1 *Formellement, une grammaire est définie à l'aide d'un quadruplet* (N, T, R, a) *où :*

1. N *représente un ensemble fini non vide de symboles non-terminaux. Ces symboles sont ceux utilisés dans les règles de production de la grammaire.*
2. T *représente un ensemble fini non vide de symboles terminaux. Ces symboles sont aussi ceux utilisés dans les règles de production de la grammaire.*
3. R *représente l'ensemble fini non vide des règles de production.*
4. $a \in N$ *est un symbole non-terminal représentant l'axiome de la grammaire.*

La grammaire formée par les règles (a) et (b) est donc définie par le quadruplet (N, T, R, a) où :

1. $N = \{$ `EXPRESSION, NATUREL` $\}$
2. $T = \{$ `+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9` $\}$
3. $R = \{(a), (b)\}$ (Par souci de concision, seules les références des règles ont été mentionnées.)
4. $a =$ `EXPRESSION`

Définition 4.2 *Le langage* $L(G)$ *d'une grammaire* G *peut lui aussi être défini de façon formelle. Ainsi :*

$$L(G) = \{x \mid x \in T^* \wedge a \Longrightarrow x\}$$

Donc, la suite de caractères x fera partie du langage $L(G)$ si et seulement si cette suite n'est composée que de symboles terminaux compris dans l'ensemble T et s'il existe une dérivation valide permettant de dériver x depuis l'axiome a.

4.2.2 Grammaire et fonctionnalités de TVL

En tant que langage formel, TVL possède sa propre grammaire formelle. Cette dernière est de type LALR, c'est-à-dire qu'elle ne contient aucune ambiguïté et qu'elle est interprétable par un analyseur LALR ("look-ahead LR"), un analyseur LR (voir Sous-section 4.2.1) optimisé. Afin de pouvoir présenter simultanément les fonctionnalités et la syntaxe de TVL, l'auteur de ce mémoire invite le lecteur à se munir de la grammaire de TVL disponible dans l'annexe B. Pour chaque fonctionnalité présentée, le lecteur peut consulter la section de la grammaire y correspondant. L'acquisition et la compréhension de la syntaxe n'en seront donc que facilitées. Toutefois, les fonctionnalités présentées ci-dessous peuvent aussi être parcourues sans avoir recours à la grammaire.

Avant de présenter toutes les fonctionnalités de TVL, il est important de d'abord présenter les règles de formation des identifiants :

- Seuls les identifiants de features doivent commencer par une majuscule.
- Les identifiants des autres éléments (attributs, types, ...) doivent eux commencer par une minuscule.

Ensuite, un identifiant peut être composé de n'importe quelle suite formée par des lettres, des chiffres et/ou le caractère "_".

Exemple global

```
1  enum socket in {LGA1156, ASB1};
2  struct dimensions {
3      real longueur;
4      real largeur;
5  }
6
7  root CarteMere
8      group oneof {
9          Asus,
10         Aopen
11     }
12
13 Asus {
14     real prix, ifin: is 350, ifout: is 0;
15     int maxRam is 24;
16 }
17
18 AOpen {
19     real prix, ifin: is 250, ifout: is 0;
20     int maxRam is 16;
21 }
22
23 CarteMere {
24     real prix;
25     int maxRam;
26     socket socketCarteMere;
27     dimensions dimensionsCarteMere;
28 }
```

Cet exemple (situé sur la page précédente) sera utilisé pour illustrer les différentes fonctionnalités de TVL. Ce modèle [3] est composé de trois features (lignes 7 à 11) : la feature racine "CarteMere" et ses deux features enfants "Asus" et "AOPen". Chacune de ces features a été déclarée une seconde fois (lignes 13 à 28) accompagnée cette fois-ci de ses attributs. Les features "Asus", "AOPen" et "CarteMere" possèdent chacune un attribut réel "prix" et un attribut entier "maxRam" (représentant la quantité maximum de mémoire ram que la carte accepte). Cependant, la feature "CarteMere" (lignes 23 à 28) possède deux attributs supplémentaires : l'attribut "socketCarteMere" défini à l'aide du type "socket" (ligne 1) et l'attribut "dimensionsCarteMere" défini à l'aide du type "dimensions" (lignes 2 à 5). Les différents éléments de ce modèle sont présentés plus en détail dans les sous-sections ci-dessous.

Composition d'un fichier TVL (Axiome "MODEL" de la grammaire)

La première règle de la grammaire indique qu'un fichier TVL peut être composé de n'importe quelle suite de déclarations de types ("`TYPE`"), de constantes ("`CONSTANT`") ou de features ("`FEATURE`"). Elle permet donc à l'utilisateur de choisir lui-même l'ordre dans lequel il veut déclarer ces trois éléments. Un utilisateur pourra d'abord préférer déclarer les types avant tout le reste alors qu'à l'inverse, un autre voudra d'abord déclarer les constantes. Ainsi, toutes les combinaisons sont possibles, ce qui permet de personnaliser au maximum l'agencement des déclarations de types, de constantes et de features. Dans l'exemple global, les types (lignes 1 à 5) ont été spécifiés avant les features.

La définition de types (Section type dans la grammaire)

En plus des types de base (entier, réel, énumératif ou booléen), TVL permet à l'utilisateur de définir ses propres types ("`TYPE`"). Ces derniers pourront être réutilisés dans n'importe quelle déclaration d'attribut. Il existe deux sortes de types :

1. Les types simples ("`SIMPLE_TYPE`") : dans la déclaration d'un tel type, il suffit de préciser le type de base (entier, réel, booléen ou énumératif), l'identifiant et éventuellement le domaine de valeur du type. Un type énumératif est un type de base spécial, il permet d'énumérer une suite d'éléments. Dans l'exemple global, le type simple "socket" (ligne 1) correspond à une énumération de sockets de processeurs. De plus, ce type est réutilisé pour définir l'attribut "socketCarteMere" (ligne 26) de la feature "CarteMere". Une carte-mère pourra donc accepter soit des processeurs avec un socket "LGA1156", soit des processeurs avec un socket "ASB1".

```
enum socket in {LGA1156, ASB1};

CarteMere {
    ...
    socket socketCarteMere;
    ...
}
```

3. Ce modèle reprend en fait un sous-ensemble des features du diagramme de la Figure 3.6 située page 40. Cependant, dans l'exemple de ce chapitre, les features ont été dotées d'attributs supplémentaires.

2. Les types structurés ("RECORD") : dans ce cas, le type peut être composé de plusieurs types simples. Dans la déclaration d'un type structuré, il faut tout d'abord indiquer le mot-clé "struct" suivi de l'identifiant du type. Ensuite, entre des accolades, il suffit de spécifier tous les types simples formant le type structuré. Dans l'exemple global , le type structuré "dimensions" (lignes 2 à 5) est composé de deux types simples : "longueur" et "largeur". Ce type est réutilisé pour définir l'attribut "dimensionsCarteMere" (ligne 27) de la feature "CarteMere".

```
struct dimensions {
    real longueur;
    real largeur;
}

CarteMere {
    ...
    dimensions dimensionsCarteMere;
    ...
}
```

La définition de constantes (Section constante dans la grammaire)

Dans certaines situations, l'utilisation de constantes ("CONSTANT") peut présenter certains avantages. Dans sa syntaxe, TVL permet d'en définir. Pour cela, il faut tout d'abord indiquer le mot-clé "const" suivi du type de base (entier, réel ou booléen) de la constante. Ensuite, il faut spécifier l'identifiant et la valeur de cette dernière. Dans l'exemple suivant, la constante "maxRam" est de type entier et représente la valeur 24.

```
const int maxRam 24;
```

La déclaration des features (Section feature de la grammaire)

Dans TVL, la déclaration des features ("FEATURE") est fort modulable. Une même feature peut être déclarée un nombre quelconque de fois. Ce mécanisme permet de, par exemple, spécifier une première fois une feature dans la structure globale du modèle, puis, de déclarer une seconde fois cette même feature, de manière individuelle et en y ajoutant alors tous ses attributs, ses contraintes... De cette manière, l'utilisateur peut avoir une vision globale du modèle, dégagée de tous les éléments additionnels (attributs, ...), ce qui facilite la compréhension.

Dans l'exemple global, les features ont été déclarées de cette façon. Les features "CarteMere", "Asus" et "AOpen" ont été spécifiées une première fois sans aucun élément (attributs, ...). Par après, elles ont de nouveau été déclarées, accompagnées cette fois de leurs attributs. Dans le corps d'une feature, il est possible de déclarer quatre éléments différents : des attributs ("ATTRIBUTE"), des contraintes ("CONSTRAINT"), des données ("DATA") et un groupe de features enfants ("FEATURE_GROUP")". Les prochaines sections étant consacrées aux trois premiers éléments, seule la déclaration des groupes de features enfants sera analysée ici.

Lors de la déclaration d'un groupe de features enfants, il faut tout d'abord indiquer le mot-clé "group". Par après, il faut préciser la cardinalité (voir Sous-section 3.2.1) de ce groupe, c'est-à-dire, le nombre minimum et maximum de

features enfants qui pourront être sélectionnées. Ceci peut se faire de deux façons différentes :

1. Via un intervalle où les bornes minimum et maximum sont précisées.
2. Via un mot-clé représentant un certain intervalle :
   (a) "`oneof`" correspondant à l'intervalle `[1..1]` et à l'opérateur logique "xor".
   (b) "`someof`" correspondant à l'intervalle `[1..*]` (où * représente le nombre de features comprises dans le groupe) et à l'opérateur logique "or".
   (c) "`allof`" correspondant à l'intervalle `[*..*]` et à l'opérateur logique "and". Dans ce cas, toutes les features du groupe doivent être sélectionnées.

Ensuite, les features enfants formant le groupe peuvent être déclarées. Chacune de ces features peut être précédée du mot-clé "`opt`", indiquant alors qu'elle est optionnelle. Dans l'exemple global, la cardinalité du groupe de features enfants de "CarteMere" a été précisée à l'aide du mot-clé "`oneof`". Cependant, l'utilisation directe de l'intervalle `[1..1]` aurait aussi été tout à fait correcte.

La déclaration des attributs (Section attributs de la grammaire)

Dans sa grammaire, TVL permet de déclarer un nombre quelconque d'attributs ("`ATTRIBUTE`") dans le corps d'une feature. Lors de la déclaration d'un attribut, l'utilisateur a le choix d'utiliser un type de base ou de réutiliser un type précédemment défini.

Dans le premier cas, il faut tout d'abord indiquer le type de l'attribut : réel, entier, booléen ou énumératif. Ensuite, il faut préciser l'identifiant de l'attribut. Ce dernier peut être suivi de la déclaration du corps de l'attribut. Dans ce corps, il est possible de préciser la valeur (précédée du mot-clé "`is`") ou le domaine de valeur (précédé du mot-clé "`in`") de l'attribut. Enfin, grâce aux mots-clés "`ifin:`" et "`ifout:`", il est aussi possible de spécifier la valeur ou le domaine de valeur de l'attribut quand la feature de celui-ci est sélectionnée ("`ifin:`") et/ou dé-sélectionnée ("`ifout:`"). Dans l'exemple global, la valeur de l'attribut "maxRam" (ligne 20) de la feature "AOpen" a été précisée à l'aide du mot-clé "`is`". A l'inverse, les valeurs de l'attribut "prix" (ligne 19) ont été précisées à l'aide de blocs conditionnels. Si la feature "AOpen" est sélectionnée, l'attribut "prix" sera égal à 250 et à 0 dans le cas contraire.

Dans le second cas, il faut premièrement indiquer l'identifiant du type utilisé. Deuxièmement, il faut spécifier l'identifiant de l'attribut. A la suite de ce dernier, il est alors possible de spécifier le corps de l'attribut. A ce moment-là, il existe deux possibilités :

1. Le type réutilisé n'est pas un type structuré. La déclaration du corps est alors identique à celle du premier cas (lorsqu'un attribut est déclaré avec un type de base). Dans l'exemple global, l'attribut "socketCarteMere" (ligne 26) de la feature "CarteMere" a été déclaré de cette façon.
2. Le type réutilisé est un type structuré. Dans ce cas, pour certains ou tous les sous-attributs de ce type structuré, l'utilisateur peut préciser l'identifiant du sous-attribut suivi du corps de ce dernier. Le corps peut être spécifié de la même façon que dans le premier cas (lorsqu'un attribut est

déclaré avec un type de base). Dans l'exemple global, l'attribut "dimensionsCarteMere" (ligne 27) de la feature "CarteMere" a été spécifié avec l'aide du type structuré "dimensions".

Dans le cas où la valeur de l'attribut est spécifiée (à l'aide du mot-clé "`is`"), il n'est pas permis d'ajouter de blocs conditionnels "`ifin:`" ou "`ifout:`". En effet, l'attribut gardera toujours cette valeur, que sa feature soit sélectionnée ou non.

La spécification des différentes valeurs (valeur de base, valeur quand la feature de l'attribut est sélectionnée, ...) d'un attribut se fait à l'aide d'expressions ("`EXPRESSION`") ou d'expressions d'ensemble ("`SET_EXPRESSION`"). Bien entendu, le type de ces dernières doit être identique au type de l'attribut auquel elles se rattachent. Dans le cas contraire, la déclaration de l'attribut n'est pas valide.

La formulation des contraintes (Section contraintes de la grammaire)

Comme expliqué plus haut, les contraintes ("`CONSTRAINT`") doivent être déclarées dans le corps des features. TVL permet de déclarer trois types de contraintes :

1. Les contraintes formées à l'aide du mot-clé "`ifin:`". Elles deviennent actives quand la feature dans laquelle elles sont déclarées est sélectionnée.
2. Les contraintes formées à l'aide du mot-clé "`ifout:`". Elles deviennent actives quand la feature dans laquelle elles sont déclarées n'est pas sélectionnée.
3. Les contraintes constamment actives. Dans ce cas, il n'y a pas besoin d'utiliser de mot-clé.

Tout comme pour les attributs, la spécification des formules des différentes contraintes se fait à l'aide d'expressions ("`EXPRESSION`").

La gestion des identifiants (Section identifiants de la grammaire)

Dans TVL, un attribut ou une feature peut être référencé de deux manières différentes. La première façon est la plus directe et la plus simple. Une feature est référencée en spécifiant son identifiant. Pour un attribut, il faut d'abord préciser l'identifiant de la feature à laquelle il appartient. Ensuite, il faut ajouter un "." puis seulement spécifier l'identifiant de l'attribut, comme dans la contrainte ci-dessous.

```
AOpen.prix == 250;
```

Pour référencer l'attribut "prix" de la feature "AOpen", on a donc d'abord spécifié l'identifiant de sa feature. Ensuite, on a rajouté le "." suivi de son identifiant.

La seconde façon permet de référencer une feature ou un attribut en spécifiant le "chemin de features" permettant d'y arriver. Ainsi, dans l'exemple global, la feature "Asus" peut être référencée à l'aide du chemin "`CarteMere.Asus`". Dans un chemin, on précise donc la suite de features (respectant la hiérarchie de features du modèle) permettant d'arriver à la feature ou à l'attribut ciblé. Tous les identifiants composant un chemin doivent être séparés entre eux par le caractère ".".

Le système des chemins a été mis en place pour permettre de référencer une feature de manière non ambiguë. En effet, dans un modèle de features, plusieurs

features peuvent posséder des identifiants identiques (dans ce cas, ces features et leurs identifiants sont dits ambigus). Il a donc fallu trouver un mécanisme capable de référencer, sans aucune ambiguïté, n'importe quelle feature d'un modèle. Dans l'exemple de la Figure 4.3, si la feature "D" est référencée en indiquant uniquement son identifiant, le référencement sera ambigu. Il pourra soit représenter la feature "D" de "B", soit représenter la feature "D" de "C". Par contre, si la feature "D" est référencée à l'aide du chemin "`B.D`" ou "`C.D`", il n'y a plus d'ambigüité possible. Grâce au mécanisme des chemins de features, n'importe quelle feature d'un modèle pourra donc toujours être référencée de manière non ambiguë.

```
root A
    group oneof {
        B group oneof {D},
        C group oneof {D}
    }
```

FIGURE 4.3 – Modèle TVL comprenant deux features ambiguës.

Enfin, TVL permet d'utiliser trois mots-clés pouvant faciliter le référencement des features :

1. "`root`" représente la feature racine. Par exemple, les deux contraintes ci-dessous sont équivalentes. Utiliser le nom de la feature racine "CarteMere" ou le mot-clé "`root`" produit le même effet.

   ```
   CarteMere.maxRam == 24;
   root.maxRam == 24;
   ```

2. "`parent`" représente la feature parente de la feature dans laquelle on se trouve actuellement. Dans l'exemple ci-dessous, les deux contraintes de la feature "AOpen" sont équivalentes. Utiliser le nom de la feature parente "CarteMere" ou le mot-clé "`parent`" produit le même effet.

   ```
   AOpen {
       CarteMere.maxRam == 24;
       parent.maxRam == 24;
   }
   ```

3. "`this`" représente la feature dans laquelle on se trouve actuellement. Dans l'exemple ci-dessous, les deux contraintes de la feature "AOpen" sont équivalentes. Utiliser le nom de la feature "AOpen" ou le mot-clé "`this`" produit le même effet.

   ```
   AOpen {
       AOpen.maxRam == 16;
       this.maxRam == 16;
   }
   ```

Ces mots-clés peuvent aussi bien être utilisés pour débuter un chemin de features que pour référencer directement une feature.

La gestion des expressions (Section expression de la grammaire)

Que ce soit dans les contraintes ou dans les déclarations d'attributs, grâce à ses expressions ("`EXPRESSION`"), TVL permet d'accomplir de nombreuses opérations. Tout d'abord, grâce aux opérateurs "`!`" (négation booléenne), "`||`" (disjonction), "`&&`"(conjonction), "`->`" (implication), "`<-`" (implication inverse) et "`<->`" (équivalence), il est possible de rédiger différentes formules booléennes, comme dans la contrainte ci-dessous :

```
(Asus -> !AOpen) && (AOpen -> !Asus);
```

Dans cet exemple, on précise que si la feature "Asus" est sélectionnée, alors la feature "AOpen" ne peut pas être sélectionnée et vice-versa. Cette contrainte est donc équivalente à la contrainte traditionnelle "exclut" ("`excludes`" dans la grammaire). Dans TVL, chaque feature est représentée par un booléen. Si la feature est sélectionnée, la valeur de ce booléen passe à "vrai", et à "faux" dans le cas contraire

Ensuite, à l'aide des opérateurs de comparaison "`==`", "`!=`", "`<=`", "`<`", "`>`" et "`>=`", TVL permet de comparer entre elles différentes expressions. Dans la contrainte ci-dessous, on précise que si la feature "Asus" est sélectionnée, son prix doit être égal à 350 ou à 0 dans le cas contraire.

```
(Asus -> Asus.prix == 350) || (!Asus -> Asus.prix == 0);
```

En utilisant les opérateurs "`+`", "`-`", "`*`" et "`/`", il est aussi possible d'effectuer différents calculs, comme dans la contrainte ci-dessous.

```
Asus.prix == (2 * 150) + 50;
```

Grâce à ses fonctions d'agrégation, TVL permet d'effectuer des opérations sur un ensemble de features, d'attributs et/ou d'expressions. Par exemple, dans la contrainte ci-dessous, la fonction "`sum`" calcule la somme des attributs "prix" des features "AOpen" et "Asus". A la place des attributs, il aurait été tout aussi possible de spécifier des expressions numériques telles que des entiers, des réels,...

```
CarteMere.prix == sum(AOpen.prix, Asus.prix);
```

De façon similaire, il est aussi possible d'utiliser des fonctions d'agrégation avec comme paramètre un attribut commun à toutes les features enfants d'une feature parente. Dans ce cas-là, il faut que chaque feature enfant possède cet attribut. De plus, il faut aussi que le type de cet attribut soit identique chez chacune des features enfants. Dans la contrainte ci-dessous, le mot-clé "`children`" combiné avec l'attribut "prix" indique qu'il faut tenir compte de l'attribut "prix" de toutes les features enfants de "CarteMere". Le calcul de cette nouvelle contrainte est donc similaire à celui de la contrainte ci-dessus.

```
CarteMere {
    this.prix == sum(children.prix);
}
```

A l'inverse, dans la contrainte située sur le haut de la page suivante, le mot-clé "`selectedchildren`" indique qu'il ne faut tenir compte que de l'attribut des features enfants ayant été sélectionnées. Donc, si la feature "Asus" est sélectionnée, le prix de la feature "CarteMere" sera égal à `sum(350)`, c'est-à-dire à 350. Dans le cas contraire, il sera égal à `sum(250)`, c'est-à-dire à 250.

```
CarteMere {
    this.prix == sum(selectedchildren.prix);
}
```

Dans la grammaire de TVL, les fonctions d'agrégation disponibles sont : "`sum`" (additionne tous ses paramètres), "`mul`" (multiplie tous ses paramètres entre eux), "`min`" (renvoit son paramètre le moins élevé), "`max`" (renvoit son paramètre le plus élevé), "`count`" (compte le nombre de ses paramètres), "`avg`" (calcule la moyenne arithmétique de ses paramètres), "`or`" (calcule une disjonction comprenant tous ses paramètres), "`and`" (calcule une conjonction comprenant tous ses paramètres) et "`xor`" (calcule une disjonction exclusive comprenant tous ses paramètres).

Finalement, TVL possède cinq autres opérateurs :

- L'opérateur "`in`" permet de tester si une expression est contenue dans un ensemble. Dans la contrainte ci-dessous, on vérifie que le nombre 5 est bien contenu dans l'intervalle `[1..9]`, ce qui est le cas.
  `5 in [1..9];`
- L'opérateur "`abs`" permet de calculer la valeur absolue d'une expression.
- L'opérateur "`?`" permet de sélectionner une seule expression parmi deux expressions. Cette sélection se fait en fonction de l'évaluation d'une première expression. Dans la contrainte ci-dessous, si la feature "Asus" est sélectionnée, le prix de cette dernière est égal à 350 ou à 0 dans le cas inverse.
  `Asus ? Asus.prix == 350 : Asus.prix == 0;`
- Les opérateurs "`excludes`" et "`requires`" correspondent respectivement aux contraintes traditionnelles "exclut" et "requiert" présentées dans le chapitre précédent. La contrainte ci-dessous signifie par exemple que les features "Asus" et "AOpen" sont incompatibles.
  `Asus excludes AOpen;`

Comme cela a été illustré dans les exemples précédents, une expression peut donc faire intervenir des features, des attributs, des valeurs basiques (entier, réel, booléen ou une valeur d'un attribut énumératif) mais aussi des ensembles. Ces derniers sont définis de la même façon que lorsqu'ils sont utilisés dans la déclaration d'un type ou d'un attribut.

Enfin, l'annexe B reprend aussi l'ensemble des règles de précédence et d'associativité des différents opérateurs. Ces règles précisent la priorité des opérateurs (par exemple, elles stipulent que l'opérateur de multiplication "`*`" est plus prioritaire que l'opérateur d'addition "`+`") et leur associativité (par exemple, l'opérateur d'addition "`+`" est associatif à gauche). Elles sont surtout utilisées durant l'analyse syntaxique.

Les blocs de données (Section données de la grammaire)

Les blocs de données ("`DATA`") permettent de stocker de l'information externe à l'intérieur des modèles TVL. Une information externe est une donnée qui n'a aucun rapport avec la syntaxe de TVL mais qui possède du sens pour un programme externe. Par exemple, dans le cas d'un logiciel de visualisation graphique de modèles TVL, ces blocs de données peuvent servir à enregistrer la manière (encadrement, couleurs, ...) dont les features sont affichées. Dans TVL,

chaque bloc de données est constitué d'un ensemble de couples (clé, valeur). Lors de la déclaration d'un bloc de données, il faut tout d'abord indiquer le mot-clé "`data`". Ensuite, entre accolades, il suffit de définir tous les couples de données. Par exemple, le bloc de données ci-dessous pourrait être rajouté à la feature "CarteMere". La clé du premier couple est "encadrementFeature" et sa valeur est "oui". Dans le second couple, la clé est "couleurIdentifiantFeature" et la valeur est "noir". Dans un diagramme, la feature "CarteMere" serait donc encadrée et son identifiant serait affiché en noir.

```
data {
    "encadrementFeature" "oui";
    "couleurIdentifiantFeature" "noir";
}
```

Structure d'un graphe acyclique orienté (Non-terminal "HIERARCHICAL_FEATURE" de la section feature de la grammaire)

Dans un modèle TVL, les features sont agencées en respectant la structure d'un graphe acyclique orienté. Chaque feature peut donc être simultanément l'enfant de plusieurs features parentes. Dans ce cas-là, une telle feature est dite partagée. Dans le modèle, le nom de cette feature devra être précédé du mot-clé "`shared`", comme dans l'exemple ci-dessous :

```
    root A
        group oneof {
            B group oneof {D},
            C group oneof {shared D}
        }
```

Dans cet exemple, la feature "D" est à la fois la feature enfant de la feature "B" et de la feature "C".

La possibilité d'inclure des fichiers TVL extérieurs

Grâce à la commande "`include()`", TVL permet d'inclure le contenu de fichiers extérieurs. Ce contenu sera inclus à l'endroit où la commande est utilisée. Ce mécanisme peut par exemple servir à fusionner différents modèles en un seul modèle global. Dans l'exemple ci-dessous, le fichier "Asus.tvl" est inclus dans le fichier "CarteMere.tvl".

Le contenu du fichier "CarteMere.tvl" :

```
root CarteMere {
    group oneof {
        include(Asus.tvl);
    }
}
```

Le contenu du fichier "Asus.tvl" :

```
Asus {
    int prix;
}
```

Le résultat de l'inclusion :

```
root CarteMere {
    group oneof {
        Asus {
            int prix;
        }
    }
}
```

Pour inclure le contenu d'un fichier, il faut donc utiliser la commande "`includ e()`" avec comme paramètre le chemin relatif ou absolu du fichier. Cette commande étant une instruction de pré-processing (elle permet de préparer le fichier TVL pour l'analyse syntaxique), elle n'est pas comprise dans la grammaire de TVL.

4.3 Avantages de TVL

Avant de commencer à énumérer les avantages de TVL, il faut souligner que ce langage n'a pas la prétention de vouloir remplacer les diagrammes de features. En proposant TVL, le but de l'équipe de LIEL est d'offrir une solution reprenant les principaux avantages de la modélisation basée sur les features, qui puisse réellement être utilisée dans le domaine industriel. De par leurs défauts (voir Section 4.1), les diagrammes de features sont rarement employés pour modéliser des problèmes industriels.

4.3.1 Modulaire

Grâce aux différents mécanismes de modularisation présentés dans la section précédente, TVL permet de personnaliser au maximum la structure d'un modèle. L'utilisateur peut, par exemple, choisir lui-même l'ordre dans lequel il désire déclarer les différents éléments (attributs, contraintes, ...). De plus, ces mécanismes peuvent aussi servir à organiser le travail en séparant les problèmes. Les features correspondant à un certain problème peuvent être isolées dans un fichier. Ainsi, chaque problème peut être traité plus facilement en se focalisant uniquement sur les features y ayant trait et en se débarrassant momentanément des éléments superflus. Par la suite, grâce à la commande d'inclusion, il suffit juste d'inclure le contenu du fichier correspondant à un problème dans le fichier TVL principal.

4.3.2 Textuel

De par la nature textuelle de TVL, les fichiers TVL pourront être exploités par un grand nombre de programmes. En effet, même avec le logiciel de traitement de texte le plus basique, il sera toujours possible de créer ou de modifier un fichier TVL. TVL est donc un langage qui ne nécessite pas d'outils dédiés. De plus, grâce à sa syntaxe proche de celle du C, les personnes ayant quelques connaissances en programmation n'auront aucun mal à le manipuler. En outre, TVL permet aussi de représenter les features et les attributs de manière plus concise que dans un diagramme. Cet avantage peut par exemple

permettre aux ingénieurs de gérer plus facilement les modèles de features. L'ensemble des documents à manipuler sera sans doute moins important que dans le cas des diagrammes de features. Enfin, vu que tous les éléments sont représentés textuellement, les attributs, les contraintes et les features peuvent être spécifiés dans le même modèle. Une contrainte additionnelle peut donc être exprimée au plus près des éléments auxquels elle se rapporte, ce qui peut donc aussi faciliter la gestion des modèles. Dans les diagrammes de features, les contraintes additionnelles ne peuvent pas être exprimées sur le graphique (sauf les contraintes "`excludes`" et "`requires`").

4.3.3 Sémantique clairement définie

Dans [8], la sémantique de TVL a été clairement définie. Grâce à cela, TVL peut facilement être comparé à d'autres langages. Il est donc possible de précisément définir les équivalences syntaxiques (les équivalences entre les instructions) entre TVL et un autre langage de features (dont la sémantique a aussi été définie). Dans ce cas, TVL peut par exemple être utilisé comme format d'échange, pour exporter des modèles de features depuis des logiciels utilisant un langage de modélisation de features fermé (dont la syntaxe et la sémantique ne sont pas connues du public, c'est souvent le cas avec les langages utilisés dans l'industrie). De plus, il peut aussi être utilisé comme format de stockage pour des outils graphiques. Grâce aux blocs de données, ces outils peuvent stocker leurs propres informations (styles de représentation des features, ...) à l'intérieur du modèle. Enfin, la définition de la sémantique permet aux modèles TVL d'être vérifiés automatiquement par des logiciels. Ces derniers pourront être capables de contrôler la validité des contraintes (e.g. qu'il n'existe pas de conflits entre les contraintes), de détecter les features mortes (e.g. les features qui ne sont utilisées dans aucun produit de la SPL), ...

Deuxième partie

Conception du parseur TVL

Chapitre 5

Motivations, exigences, fonctionnalités et architecture

Après la présentation du contexte de la problématique, ce cinquième chapitre va permettre d'introduire la problématique elle-même et de présenter la solution qui y est apportée. La première section sera donc consacrée à la problématique, c'est-à-dire, à l'absence d'outils automatisant le contrôle et l'analyse de modèles TVL. Ensuite, la seconde section permettra de présenter les exigences requises pour la solution, autrement dit, pour le parseur TVL. Par après, la troisième section listera l'ensemble des fonctionnalités de ce parseur tandis que la dernière section exposera l'architecture du logiciel et permettra de justifier certains choix de design.

Pour information, le parseur TVL peut être téléchargé à l'adresse : `http://www.info.fundp.ac.be/~acs/tvl/`, dans la partie "Implémentation". Toutes les instructions nécessaires à son utilisation sont aussi disponibles sur cette page web.

5.1 Motivations

Comme expliqué dans le chapitre précédent, TVL est un langage qui a été développé afin de répondre aux besoins du monde industriel. Cependant, présenté ainsi, TVL ne peut pas être pleinement exploité. N'importe quelle entreprise pourra développer un modèle TVL mais au moment de contrôler et d'analyser ce dernier, les choses se compliqueront :

- Contrôler manuellement (sans l'aide d'un logiciel) un modèle composé de centaines de features, de centaines d'attributs, de centaines de contraintes, ... peut être une tâche complexe pouvant nécessiter de nombreuses ressources (humaines, temps, ...).
- Calculer manuellement des opérations par rapport à ce genre de modèles peut être impossible. En effet, si un développeur veut vérifier la satisfiabilité du modèle, il devra lui-même calculer toutes les combinaisons possibles de features et d'attributs respectant l'ensemble des contraintes

du modèle (structure, cardinalités et contraintes additionnelles). Vu que le modèle peut être composé de plusieurs centaines d'éléments, cette tâche s'annonce colossale voire impossible.

Dans ce contexte, les avantages de TVL risqueraient d'être éclipsés par les inconvénients liés au contrôle des modèles. Les entreprises pourraient alors ne trouver aucun intérêt à utiliser le langage. En effet, TVL serait uniquement utile au moment de créer le modèle. Par après, vu l'**absence de logiciels**, le contrôle et l'analyse des modèles risqueraient de nécessiter trop de ressources que pour "rentabiliser" l'utilisation de TVL.

Dès lors, **afin de combler ce vide** et de donner un aperçu des avantages pratiques de TVL, **un logiciel automatisant le contrôle et l'analyse des modèles doit être développé**. Les exigences requises pour ce logiciel sont exposées dans la section suivante.

5.2 Exigences

Afin de proposer une solution réellement adaptée à la problématique, l'outil doit respecter les cinq exigences présentées ci-dessous.

A. Contrôle de la syntaxe et de la sémantique d'un modèle TVL

Le logiciel doit être capable de contrôler aussi bien la structure que le contenu d'un modèle TVL. Il devra par exemple vérifier que le modèle décrit respecte bien l'ensemble des règles de la grammaire de TVL ou que les contraintes sont correctement formées (utilisation correcte des opérateurs, des identifiants, ...).

B. Génération automatique de la forme normalisée d'un modèle TVL

Dans [8], les auteurs présentent une liste de transformations permettant de générer la forme normalisée d'un modèle TVL. Une telle forme est créée à l'aide d'un sous-ensemble des constructeurs de TVL. Son principal avantage est de pouvoir être réutilisée afin d'être soumise à différents types de solveurs. Cependant, transformer un modèle manuellement demande beaucoup de concentration et de temps. Dès lors, afin de simplifier le travail des industriels, l'outil doit être capable de générer automatiquement la forme normalisée d'un modèle TVL.

C. Contrôle de la satisfiabilité d'un modèle TVL

Dans le but de s'assurer de la validité de son modèle, un développeur pourra vouloir en tester la satisfiabilité. Un modèle est dit satisfiable s'il existe au moins une combinaison de features et d'attributs respectant l'ensemble des contraintes (structure, cardinalités et contraintes additionnelles) de ce modèle. A l'aide d'un solveur, l'outil doit donc être en mesure de déterminer si oui ou non un modèle est satisfiable.

D. Possibilité d'utiliser le logiciel comme une API

Afin de s'adapter correctement aux différents environnements des entreprises, en plus de pouvoir être employé indépendamment de tout autre logiciel, l'outil

doit aussi pouvoir être utilisé comme une API. De cette manière, les ingénieurs d'une entreprise pourront par exemple continuer à utiliser leurs propres logiciels tout en interagissant avec l'outil.

E. Portabilité

Dans les entreprises, différents systèmes d'exploitation sont employés. Afin de pouvoir être utilisé dans n'importe quelle société, l'outil doit être portable (utilisable sous la plupart des systèmes d'exploitation actuels).

5.3 Fonctionnalités

Suite aux faits exposés dans la Section 5.1, un logiciel nommé "parseur TVL" a été développé. Dans le but de respecter les exigences "A", "B" et "C", ce logiciel a été doté de trois fonctionnalités majeures :

L'analyse syntaxique et sémantique d'un modèle TVL

Afin de pouvoir respecter l'exigence "A", le logiciel est capable de vérifier syntaxiquement et sémantiquement un modèle TVL. Pour cela, il procède à toute une série de vérifications permettant de par exemple s'assurer que le modèle ne contient pas de cycles. Si aucune erreur n'est détectée, le logiciel produit les trois tables des symboles correspondant au modèle. La partie analyse du parseur est abordée dans le Chapitre 6.

La génération de la forme normalisée d'un modèle

L'exigence "B" impose au parseur de pouvoir générer automatiquement la forme normalisée d'un modèle. A partir d'un modèle contrôlé syntaxiquement et sémantiquement correct, le logiciel peut donc générer la forme normale de ce modèle. Le Chapitre 7 traite de la génération de la forme normale au sein du parseur.

Le contrôle de la satisfiabilité d'un modèle

L'exigence "C" contraint le logiciel à pouvoir vérifier la satisfiabilité d'un modèle. Pour cela, il soumet la forme normale d'un modèle, moyennant quelques transformations, à un solveur. Actuellement, le solveur utilisé est un solveur de type SAT. Ceci est justifié par le fait que de tels solveurs permettent de manipuler des modèles pouvant contenir jusqu'à dix mille features [18].

Lors de la soumission d'un modèle TVL normalisé à un solveur SAT, le parseur procède en deux étapes. Durant la première, il génère la forme booléenne du modèle normalisé. Un modèle TVL respectant cette forme ne peut par exemple contenir que des attributs booléens. Ceci est dû au fait qu'un solveur SAT n'accepte que des problèmes uniquement composés de variables booléennes. La forme booléenne d'un modèle est contrôlable par le parseur et peut être réutilisée afin d'être soumise à un autre solveur SAT que celui actuellement employé par le logiciel.

Ensuite, durant la seconde étape, le logiciel génère le problème SAT correspondant au modèle booléen. A partir de là, il est possible de calculer différentes

opérations par rapport au modèle. Il est notamment possible de déterminer si oui ou non le modèle est satisfiable.

La génération de la forme booléenne et la transformation de cette dernière en un problème SAT sont abordées dans le Chapitre 8.

5.4 Architecture

Le parseur TVL a été implémenté à l'aide du langage orienté objet Java [1]. Ceci permet de respecter l'exigence "E". En effet, les programmes utilisant ce langage peuvent être exécutés sous la plupart des systèmes d'exploitation actuels. La seule pré-condition est d'installer l'environnement Java compatible avec le système d'exploitation sous lequel le parseur TVL est utilisé.

La structure globale du parseur est composée de deux classes principales et de cinq packages : "Parser", "SyntaxTree", "SymbolTable", "Exceptions" et "Util" (comme l'illustre la Figure 5.1). Ces deux classes sont les points d'entrées du parseur et permettent de l'utiliser soit comme un logiciel indépendant, soit comme une librairie. Cette décomposition en deux classes permet de respecter l'exigence "D", imposant de pouvoir utiliser le parseur TVL en tant que librairie.

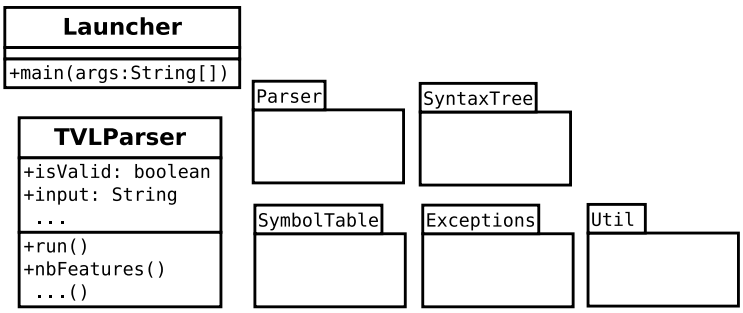

FIGURE 5.1 – Structure globale du parseur TVL.

Classe principale "Launcher"

Cette première classe permet d'utiliser directement le parseur comme un programme à part entière et indépendamment de tout autre logiciel. Grâce à cette classe, il est possible de lancer l'analyse d'un fichier via l'invité de commandes shell. Par exemple, pour lancer l'analyse du fichier "voiture.tvl", il suffit de taper la commande :

```
java -jar TVLLibrary.jar -s voiture.tvl
```

où "TVLibrary.jar" représente l'archive contenant le parseur. L'unique méthode de la classe "Launcher" fait appel aux méthodes contenues dans la classe "TVLParser".

1. Site web officiel de Java : `http://www.java.com/fr`

Classe principale "TVLParser"

Cette seconde classe permet d'utiliser le parseur TVL comme une librairie. N'importe quel programme peut alors faire appel aux méthodes implémentées dans cette classe afin d'analyser un fichier TVL. Ces méthodes permettent :

- D'analyser syntaxiquement et sémantiquement un fichier TVL (voir Chapitre 6).
- De générer la forme normalisée d'un modèle TVL (voir Chapitre 7).
- De générer la forme booléenne d'un modèle TVL (voir Chapitre 8).
- De calculer, à l'aide d'un solveur SAT, différentes opérations concernant la forme booléenne (voir Chapitre 8).
- D'obtenir des informations concernant l'analyse et les différentes formes d'un modèle TVL. Il est par exemple possible de connaître le nombre de features contenues dans un modèle, le nombre d'attributs booléens qui ont été déclarés, ...

Package "Parser"

Ce package contient les trois classes nécessaires à l'analyse lexicale et syntaxique. Ici, l'analyseur lexical a été généré automatiquement grâce à JFlex [2], l'équivalent Java de Lex (un générateur d'analyseurs lexicaux, voir première section du chapitre suivant). De son côté, l'analyseur syntaxique a lui aussi été généré automatiquement grâce à CUP [3], l'équivalent Java de Yacc (un générateur d'analyseurs syntaxiques, voir première section du chapitre suivant).

Package "SyntaxTree"

Ce package inclut l'ensemble des classes nécessaires à la construction des arbres syntaxiques produits par l'analyseur syntaxique. A quelques exceptions près, il existe une classe pour chaque symbole non-terminal de la grammaire et pour chaque alternative de la règle de production du non-terminal "EXPRESSION". Par exemple :

- La classe "Feature" correspond au symbole non-terminal "`FEATURE`".
- La classe "Attribute" correspond au symbole non-terminal "`ATTRIBUTE`".
- La classe "AndExpression" correspond à l'alternative "`EXPRESSION "&&" EXPRESSION`"
- La classe "EqualsExpression" correspond à l'alternative "`EXPRESSION "==" EXPRESSION`"
- La classe "LongIDExpression" correspond à l'alternative "`LONGID`"
- ...

La représentation virtuelle d'un arbre syntaxique se fait donc à l'aide d'objets Java. Ces derniers sont reliés entre eux par des références et certains peuvent contenir des informations (la représentation virtuelle est donc quasi identique à la représentation réelle, voir Sous-section 4.2.1). Par exemple, un objet "LongIDExpression" contient la chaîne de caractères correspondant au "`LONGID`" qu'il représente. En outre, toutes les classes représentant une alternative de la règle "`EXPRESSION`" :

- Non-compatibles avec la forme booléenne (voir Chapitre 8) implémentent l'interface "Expression". Cette dernière contient deux méthodes permettant de renvoyer le type et la forme normale d'une expression.

2. Page web officielle de JFlex : `http://jflex.de/`
3. Page web officielle de CUP : `http://www2.cs.tum.edu/projects/cup/`

– Compatibles avec la forme booléenne (voir Chapitre 8) implémentent l'interface "BooleanExpression" (qui elle-même étend l'interface "Expression"). "BooleanExpression" contient une méthode permettant de transformer une expression booléenne complexe en une expression booléenne simple. En outre, elle inclut aussi les méthodes permettant de générer la forme normale conjonctive d'une expression booléenne.

Chaque méthode citée ci-dessus est donc réalisée par les classes implémentant directement ou indirectement l'interface de cette méthode. L'implémentation de ces interfaces permet de faciliter la gestion des expressions. En effet, par exemple, lors de la génération de la forme normalisée, chaque expression "s'auto-transforme" elle et toutes les sous-expressions qu'elle contient en une expression normalisée. Pour générer la forme normalisée de l'expression E_1 ci-dessous, il faut faire appel à sa méthode de normalisation. Dès lors, E_1 va aussi faire appel aux méthodes de normalisation de ses deux sous-expressions E_2 et E_3 afin de les normaliser. Si ces deux sous-expressions sont aussi composées d'autres sous-expressions, elles vont aussi faire appel aux méthodes de normalisation de ces dernières afin de les normaliser et ainsi de suite. De cette manière, la forme normale de n'importe quelle expression, même la plus complexe, peut être facilement générée.

$$E_1 = "E_2 \text{ + } E_3"$$

Package "SymbolTable"

Ce package contient toutes les classes utilisées pour construire les trois tables des symboles. Ces dernières sont abordées en détails dans la seconde section du chapitre suivant.

Package "Exceptions"

Ce package inclut l'ensemble des classes des exceptions pouvant être générées durant l'analyse d'un modèle. Chaque exception correspond à un problème spécifique :

– L'exception "AmbiguousReferenceException" est renvoyée quand un chemin de features est ambigu.
– L'exception "CycleFoundException" est renvoyée si un cycle est détecté dans le graphe.
– L'exception "ParsingException" est renvoyée quand une erreur est détectée durant l'analyse syntaxique d'un fichier TVL.
– ...

Package "Util"

Ce package contient l'ensemble des classes n'ayant aucun rapport avec les thèmes des packages présentés précédemment. La classe :

– "BooleanForm" permet de générer la forme booléenne d'un modèle normalisé (voir Chapitre 8)
– "NormalForm" permet de générer la forme normale d'un modèle (voir Chapitre 7).

– "FeatureStack" représente une pile de features. Elle est utilisée durant la construction des tables des symboles.
– "IDGenerator" permet de générer des identifiants numériques uniques pour chaque feature ou attribut de la forme booléenne.
– "Solver" représente le solveur SAT utilisé pour analyser la forme booléenne d'un modèle (voir Chapitre 8).
– "Util" contient différentes méthodes permettant d'accomplir des opérations élémentaires. Par exemple, vérifier qu'une chaîne de caractères commence bien par une majuscule.

Chapitre 6

Analyse syntaxique et sémantique

Suite à la présentation globale du parseur TVL, le présent chapitre va permettre de présenter en détails sa première fonctionnalité : le contrôle de la syntaxe et de la sémantique d'un fichier TVL. Lors de la vérification d'un modèle, le parseur vérifiera donc aussi bien la structure que le contenu, il contrôlera par exemple que le modèle ne contient pas de cycles. De plus, à la fin de cette analyse, le parseur produira trois tables de symboles qui pourront être réutilisées afin de générer la forme normale du modèle (voir chapitre suivant). Dans ce chapitre, la première section permettra tout d'abord de rappeler les étapes-clés de l'analyse d'un fichier tandis que la deuxième présentera les structures des trois tables des symboles générées durant l'analyse sémantique. Enfin, la troisième section s'attardera sur le fonctionnement global de l'analyse d'un fichier au sein du parseur TVL.

6.1 Rappel : les phases-clés de la compilation

Même si le logiciel développé n'est pas un compilateur, les premières phases d'analyse d'un fichier TVL sont similaires à celles qu'un compilateur pratique pour analyser un fichier. La compilation d'un fichier peut être décomposée en plusieurs étapes.

La première étape est l'analyse lexicale. Lors de cette dernière, le contenu du fichier à compiler est vu comme une suite de caractères. L'analyse lexicale va donc avoir pour but de regrouper ces caractères entre eux afin de former des tokens. Les règles de formation de ces derniers sont simples, l'ensemble des tokens qu'il est possible de former correspond à l'ensemble des symboles terminaux de la grammaire. Chaque token doit donc être formé de manière à correspondre à un symbole terminal. L'output de l'analyse lexicale est donc une suite de symboles terminaux. Dans la plupart des cas, le compilateur ne s'occupe pas lui-même de l'analyse lexicale, il fait souvent appel à un programme externe appelé analyseur lexical. Le générateur d'analyseurs lexicaux le plus connu est sans doute Lex[1]. A partir de l'ensemble des symboles terminaux

1. Vu l'impossibilité de trouver la page web officielle de Lex, l'auteur cite ici comme réfé-

d'une grammaire, un tel programme est en mesure de générer automatiquement un analyseur lexical capable de former des tokens correspondant à ces symboles terminaux.

FIGURE 6.1 – Représentation imagée d'une analyse lexicale.

Dans la Figure 6.1, l'input de l'analyse lexicale est la suite de caractères : "`2,3,5, ,+, ,7,2`". Dans l'output, les caractères ont été regroupés pour former une suite de trois symboles terminaux : "`235,+,72`". Dans cet exemple et dans l'exemple ci-dessous (Figure 6.2), la grammaire utilisée est celle présentée dans la Sous-section 4.2.1 du chapitre précédent.

La deuxième étape est l'analyse syntaxique. Durant celle-ci, il faut vérifier qu'il existe bien une dérivation permettant de générer la suite de symboles transmise par l'analyse lexicale. Pour une suite de symboles reçue, l'analyseur va donc reconstruire la dérivation permettant de générer cette suite. Dans le cas où cette dernière est reconnue (e.g. il existe une dérivation correspondant à cette suite), l'analyse syntaxique produit l'arbre syntaxique correspondant à sa dérivation. Tout comme pour l'analyse lexicale, un compilateur s'occupe rarement lui-même de l'analyse syntaxique, il requiert souvent les services d'un programme externe appelé analyseur syntaxique. Le générateur d'analyseurs syntaxiques le plus connu est sans doute Yacc [2]. A partir des règles d'une grammaire, un tel programme peut générer automatiquement un analyseur syntaxique capable de reconnaître les mots de cette grammaire.

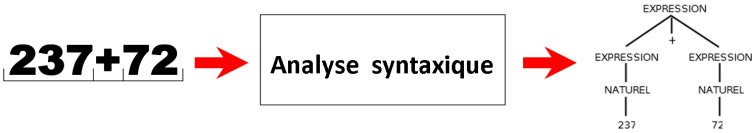

FIGURE 6.2 – Représentation imagée d'une analyse syntaxique.

Dans la Figure 6.2, l'output de l'analyse syntaxique est l'arbre syntaxique correspondant à la dérivation de la chaîne de symboles "`235,+,72`".

La troisième étape est l'analyse sémantique. Durant les deux premières étapes, seule la forme du contenu du fichier a été contrôlée. Sa signification, son sens n'ont pas encore été vérifiés. Par exemple, après une analyse lexicale et syntaxique, la phrase "Romain a mangé un bâtiment" sera considérée comme correcte. En effet, la structure et les accords respectent les règles de la grammaire française. Cependant, si l'on fait maintenant attention au sens, la phrase n'est plus valide. Le rôle de l'analyse sémantique est donc de contrôler le sens du contenu du fichier. Pour cela, elle va se servir de l'arbre syntaxique produit par

rence la page web de Flex, l'équivalent sous licence GNU de Lex : `http://flex.sourceforge.net/`

2. Tout comme pour Lex, il n'a pas été possible de trouver la page web officielle de Yacc. L'auteur cite ici comme référence la page web de Bison, l'équivalent sous licence GNU de Yacc : `http://www.gnu.org/software/bison/`

l'analyse syntaxique. Parallèlement, elle va aussi avoir besoin de l'ensemble des règles concernant la sémantique du langage (décrites formellement ou informellement). Cet ensemble va par exemple contenir les règles portant sur l'utilisation des identifiants des variables, des fonctions, ... Une de ces règles pourra par exemple stipuler que les identifiants des fonctions doivent être uniques. Dès lors, si durant l'analyse sémantique, le compilateur rencontre deux fonctions ayant un identifiant identique, il renvoit une erreur et l'analyse échoue.

L'output de l'analyse sémantique est une table des symboles. Cette dernière va contenir l'ensemble des éléments identifiés (e.g. les éléments possédant un identifiant. Par exemple : les variables, les constantes, ...) déclarés dans le contenu du fichier. Chacun de ces éléments sera caractérisé par une liste d'attributs (si c'est une variable, cela pourra par exemple être son type et sa valeur). Pour construire cette table, il est souvent nécessaire de parcourir plusieurs fois l'arbre syntaxique.

A la fin de cette troisième étape, il n'existe plus de similitudes entre le parseur TVL et un compilateur.

6.2 Structure des tables des symboles

L'analyse sémantique est une étape-clé de l'analyse d'un fichier TVL. Durant cette dernière, le parseur TVL accomplit deux tâches importantes. Premièrement, il vérifie que le modèle respecte bien tout un ensemble de règles sémantiques et syntaxiques (voir annexe D). Deuxièmement, il construit les trois tables des symboles : la table des constantes, la table des types et la table des features. L'objectif de ces tables est de centraliser toutes les informations concernant une même catégorie d'éléments. Ceci permettra, d'une part, de pouvoir les consulter afin d'obtenir divers renseignements (concernant les contraintes, les blocs conditionnels des attributs,...) et d'autre part, de pouvoir les réutiliser afin de générer la forme normalisée d'un modèle.

Chacune de ces trois tables est structurée comme un dictionnaire de données. Un tel dictionnaire est constitué d'un ensemble de couples (clé, objet). Pour pouvoir accéder à un objet, il faut préciser sa clé. Dans un dictionnaire, chacune de ces clés doit être unique. En ce qui concerne :

- La table des features, les couples sont du type (ID, feature) où ID représente l'identifiant de la feature.
- La table des types, les couples sont de la forme (ID, type) où ID représente l'identifiant du type.
- La table des constantes, les couples sont du type (ID, constante) où ID représente l'identifiant de la constante.

Dans tous les cas, l'accès à un symbole (feature, type ou constante) se fait toujours via son identifiant.

La table des features

Comme son nom l'indique, cette table va contenir toutes les informations concernant les features. Une telle table sera représentée par un objet de la classe "FeaturesSymbolTable". Cette dernière inclut tous les éléments caractérisant une table (le nombre de features contenues, ...) et toutes les méthodes permettant

de créer et d'explorer une table. De plus, chaque élément (feature, attribut, ...) enregistré dans cette table est lui-même aussi représenté par un objet java spécifique. Pour chacun de ces éléments, la classe lui correspondant contient différents attributs et méthodes permettant de le caractériser et de le manipuler. Ainsi :

- **Une feature** est représentée par un objet de la classe "FeatureSymbol". Elle est caractérisé par son identifiant, la liste de ses attributs, la liste de ses features enfants, la liste de ses features parentes, la liste des contraintes spécifiées dans sa/ses déclaration(s), ...
- **Un attribut** est représenté par un objet de la classe "AttributeSymbol". Il est caractérisé par son identifiant, son type, son domaine de valeurs, ... Les attributs structurés sont eux représentés par un objet de la classe "RecordSymbol" (cette classe étend "AttributeSymbol"). L'utilisation d'une classe spécifique était nécessaire pour pouvoir modéliser la structure de tels attributs.
- **Une contrainte** est représentée par un objet de la classe "ConstraintSymbol". Dans cette dernière, l'expression de la contrainte est toujours construite à l'aide des classes du package "SyntaxTree" (voir Section 5.4).
- **Le domaine de valeurs** d'un attribut peut être représenté de deux façons. Si c'est un intervalle, il est représenté par un objet de la classe "IntervalSetSymbol" tandis que si c'est une énumération, il est représenté par un objet de la classe "EnumSetSymbol". Ces deux classes implémentent l'interface "SetSymbol".

Dans la Figure 6.3 (située sur la page suivante), à chaque feature du modèle correspond un emplacement (sauf pour les features ambiguës, voir paragraphe suivant) de la table. Ce dernier contient la "FeatureSymbol" correspondant à la feature. Les éléments en gras et en italique représentent des références vers les objets java les incarnant (par souci de concision, à part pour l'attribut "prix", pour la feature "Voiture" et pour les features ambiguës "Peugeot", les flèches matérialisant ces références n'ont pas été représentées). Par exemple, dans la "FeatureSymbol" Véhicule, l'attribut "prix" est une référence vers l'"AttributeSymbol" le représentant. De manière similaire, la feature "Voiture" est une référence vers la "FeatureSymbol" Voiture de la deuxième case. Ce système de référence s'appuie sur le fait qu'à tout moment, un élément (feature, attribut, ...) est représenté par un et un seul objet. Grâce à ce mécanisme, si des modifications doivent par exemple être apportées à la feature "Voiture", elles ne devront l'être que sur l'unique "FeatureSymbol" la représentant.

Toujours dans le modèle de la Figure 6.3, deux features sont déclarées avec un identifiant identique : "Peugeot". Ces derniers et leur feature sont donc dits ambigus et dans la table des features, l'emplacement correspondant à cet identifiant est vide (il est grisé dans la figure). Toute tentative d'y accéder se soldera par une exception. Des features ambiguës ne sont pas directement enregistrées dans la table, elles le sont indirectement par l'intermédiaire de la/des références(s) avec leur(s) feature(s) parente(s). Ainsi, par exemple, la feature "Peugeot" de "Voiture" est enregistrée dans la table grâce à la référence de sa feature parente. Pour accéder à cette feature, il sera donc obligatoire de passer par sa feature parente en utilisant le chemin "Voiture.Peugeot" ou "Véhicule.Voiture.Peugeot".

Dans une table des features, chaque "FeatureSymbol" contient donc les références vers ses features enfants. De manière similaire, les "FeatureSymbol" de

```
root Véhicule {
    int prix;
    group oneOf {
        Voiture {group oneof {Peugeot, Porsche}},
        Camion  {group oneof {Peugeot, Renault}}
    }
}
```

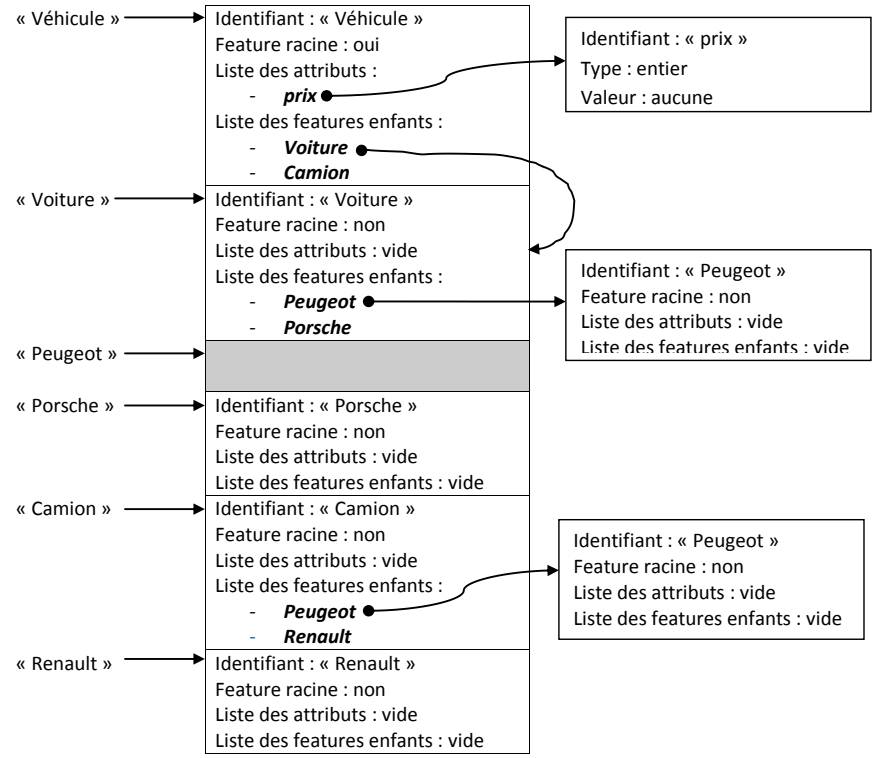

FIGURE 6.3 – Exemple d'une table de features.

ces enfants contiennent aussi les références vers leurs propres features enfants et ainsi de suite. A partir d'une "FeatureSymbol", via les références, il sera toujours possible d'accéder à toutes les features "descendantes" de cette feature. Donc, à partir de la "FeatureSymbol" de la feature racine du modèle, il est possible d'accéder à n'importe quelle feature du modèle. Il est donc primordial de contrôler que l'identifiant de la feature racine est bien unique au risque de ne plus pouvoir accéder à aucune feature du modèle.

La table des types

Une telle table contient l'ensemble des types qui ont été déclarés dans le modèle. Au niveau de l'implémentation, elle est représentée par un objet de la classe "TypesSymbolTable". Selon sa catégorie, chaque type peut être représenté de deux façons différentes :

- Un type simple est représenté par un objet de la classe "AttributeSymbol". Il peut être caractérisé par son identifiant, son type de base (entier, booléen, réel ou énumératif) et son domaine de valeurs.
- Un type structuré est représenté par un objet de la classe "RecordSymbol". Il est caractérisé par son identifiant et l'ensemble des types simples qu'il contient.

Pour représenter les types, vu les similarités importantes entre ces derniers et les attributs, il a été jugé plus simple de réutiliser les classes "AttributeSymbol" et "RecordSymbol". En outre, les domaines de valeurs ont de nouveau été représentés en utilisant les classes "EnumSetSymbol" et "IntervallSetSymbol".

Dans la table des types de la Figure 6.4, pour chaque type déclaré, il existe un emplacement dans la table. Ce dernier contient le symbole ("AttributeSymbol" ou "RecordSymbol") représentant le type. Ici, les éléments en gras et en italique représentent de nouveau des références vers les objets java les incarnant. Par exemple, le type simple "largeur" est en fait une référence vers l'"AttributeSymbol" le représentant.

```
struct dimension {
    real largeur;
    real longueur;
}
int nbrPlaces;
```

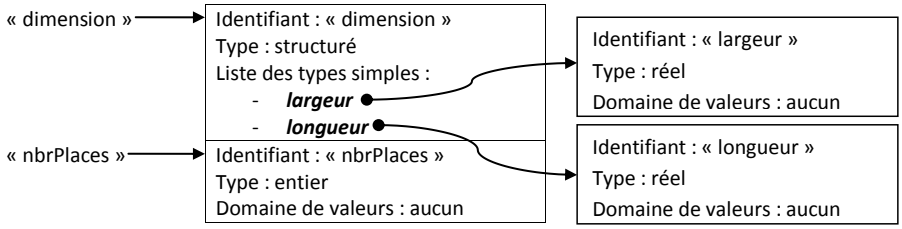

FIGURE 6.4 – Exemple d'une table des types.

La table des constantes

Cette table est sans doute celle dont la structure est la plus simple. Elle ne contient aucune référence vers des éléments extérieurs ("FeatureSymbol", ...). Une telle table est représentée par un objet de la classe "ConstantsSymbolTable". Chaque constante y est incarnée par un objet de la classe "ConstantSymbol". Cette dernière permet de caractériser chaque constante grâce à son identifiant, son type et sa valeur.

Dans la table des constantes de la Figure 6.5, pour chaque constante déclarée, il existe un emplacement dans la table. Ce dernier contient la "ConstantSymbol" représentant la constante.

```
const int vitesseMax 250;
const real prixMin 100000;
```

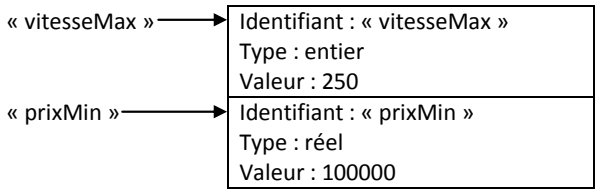

FIGURE 6.5 – Exemple d'une table des constantes.

6.3 Fonctionnement global de l'analyse d'un fichier TVL

Après la présentation des différentes structures des tables des symboles, il est maintenant possible de s'intéresser au fonctionnement global de l'analyse pratiquée par le parseur TVL. Comme illustré sur le schéma de la Figure 6.6, l'analyse est composée de trois phases.

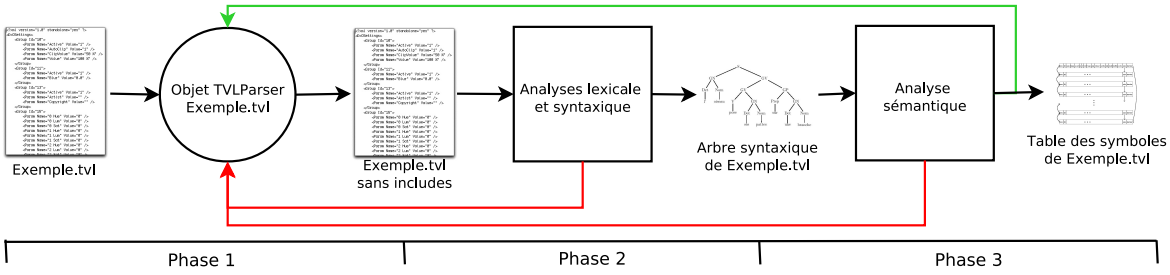

FIGURE 6.6 – Schéma illustrant le fonctionnement global de l'analyse d'un fichier TVL.

Durant la première phase, il faut soumettre le fichier TVL à analyser au parseur TVL. Pour cela, il faut créer un objet "TVLParser" (voir Section 5.4) avec comme paramètre le chemin du fichier à analyser. La première chose que le parseur TVL va vérifier, c'est si ce fichier contient des "`includes`". Si c'est le cas, il va les résoudre un à un et reconstituer un fichier unique. L'objet "TVLParser" est primordial, c'est lui qui va permettre de lancer toutes les opérations (analyse, génération de la forme normalisée, ...) concernant le modèle contenu dans le fichier TVL. De plus, si une exception est rencontrée durant l'une des phases (flèches rouges sur le schéma), c'est à lui qu'elle sera transmise. Ici, le principe est que, quel que soit le résultat de l'analyse (échec ou succès), l'objet "TVLParser" reste disponible. De cette manière, l'utilisateur pourra par exemple toujours

s'informer des causes de l'échec d'une analyse. Pour lancer l'analyse même du fichier TVL, il faut utiliser la méthode "run()" de l'objet "TVLParser". A ce moment-là, le parseur TVL passe à deuxième phase.

Durant cette phase, le parseur TVL va soumettre le fichier TVL aux analyseurs lexicaux et syntaxiques. Si le fichier est jugé comme syntaxiquement correct, l'analyseur syntaxique produit l'arbre syntaxique correspondant au modèle contenu dans le fichier. Dans le cas contraire, comme expliqué dans le paragraphe précédent, une exception est transmise à l'objet "TVLParser" et l'analyse s'arrête car elle a échoué.

Dans le cas où les analyses lexicales et syntaxiques se sont bien déroulées, le parseur TVL passe à la phase trois, l'analyse sémantique. Cette dernière phase est elle-même subdivisée en trois étapes, comme illustré sur la Figure 6.7. Durant ces trois étapes, le logiciel va vérifier que différentes règles syntaxiques et sémantiques sont bien respectées. Ces règles sont divisées en trois catégories. La première contient l'ensemble des règles concernant les identifiants. Par exemple, une de ces règles stipule que l'identifiant de la feature racine doit être unique. Ensuite, la seconde catégorie reprend toutes les règles ayant trait à la structure du modèle. Une des règles de cette catégorie interdit par exemple la présence de cycles dans le graphe. Enfin, la dernière catégorie inclut l'ensemble des règles concernant le type checking. Par exemple, elle contient la règle précisant que toutes les contraintes doivent être de type booléen. Une liste non exhaustive des règles de ces trois catégories est disponible dans l'annexe D.

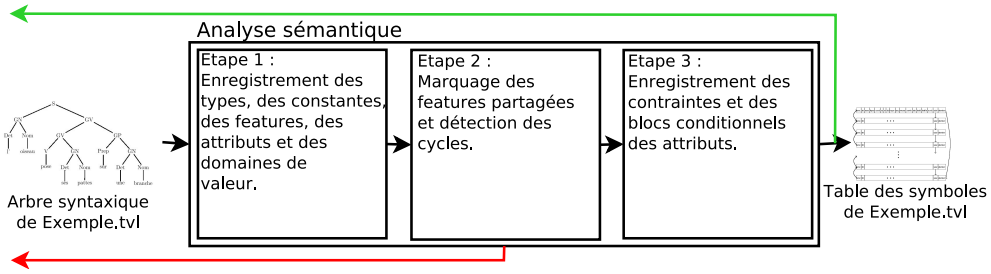

FIGURE 6.7 – Schéma illustrant le fonctionnement de l'analyse sémantique.

Cette subdivision en plusieurs étapes provient du fait que certaines vérifications ne peuvent se faire sans avoir au préalable rempli certaines conditions. Il est par exemple impossible de vérifier les contraintes et les éléments (attributs, features, ...) y entrant en jeu sans avoir auparavant enregistré ces éléments dans les différentes tables des symboles.

Durant la première phase de l'analyse sémantique, le parseur TVL va d'abord récupérer tous les types contenus dans l'arbre syntaxique afin de créer la table des types. A ce moment-là, avant d'enregistrer chaque type, il opérera toute une série de vérifications (il vérifiera qu'il n'existe déjà pas un type possédant le même identifiant, ...). Après, le logiciel va faire de même pour les constantes afin de construire la table des constantes.

Ensuite, toujours durant la première étape, le parseur TVL va récupérer toutes les déclarations de features contenues dans l'arbre syntaxique. Pour chaque déclaration, il va :

– Enregistrer la feature et ses caractéristiques (racine ou optionnelle) dans la table des features. Pour cela, une nouvelle "FeatureSymbol", correspondant

à la feature, va donc être créée dans la table. Dans le cas où la feature est déjà enregistrée dans la table, le parseur TVL passe directement au pas suivant.
- Enregistrer, s'il y en a, les attributs de la feature dans la table des features. Chaque attribut et son domaine de valeur vont donc être rattachés à la "FeatureSymbol" représentant la feature. Lors de la sauvegarde d'un attribut, le logiciel procède à tout un ensemble de tests (il vérifie s'il n'existe déjà pas un attribut possédant le même identifiant, que le domaine de valeurs de l'attribut concorde bien avec son type, ...). Dans le cas où l'attribut possède un type défini par l'utilisateur, le parseur TVL fait le lien avec ce type et vérifie si ce dernier est utilisé correctement.
- Enregistrer, s'il y en a, les features enfants de la feature dans la table des features. Chaque feature enfant va être rattachée à la "FeatureSymbol" correspondant à la feature. En outre, la déclaration de chaque feature enfant va être analysée tout comme l'a été la déclaration de sa feature parente. Lors de cette étape, les features enfants marquées "`shared`" ne sont pas traitées car elles le seront durant l'étape suivante.

Lorsque le parseur TVL a terminé d'analyser une déclaration de feature, il passe à la suivante jusqu'à ce qu'il les ait toutes parcourues.

A la seconde étape, le parseur va de nouveau parcourir les déclarations de features mais en ne faisant attention cette fois-ci qu'aux features partagées. Chaque fois qu'il trouvera une feature marquée "`shared`", il créera le lien (dans la table des features) entre cette feature et sa nouvelle feature parente. Par après, le logiciel vérifiera que les nouveaux liens n'ont pas créé de cycles dans le graphe. A la fin des deux premières étapes, la table des features n'est que partiellement construite, elle ne le sera complètement qu'à la fin de la troisième étape.

Enfin, lors de la troisième étape, le parseur TVL va explorer une dernière fois l'arbre syntaxique en ne s'intéressant qu'aux attributs et aux contraintes. Pour chaque attribut, il va vérifier son/ses blocs(s) conditionnel(s). Il va par exemple contrôler que l'expression spécifiée dans un des blocs est bien du même type que l'attribut, ... Chaque fois qu'un bloc conditionnel sera considéré comme correct, il sera rajouté à l'attribut lui correspondant dans la table des features. Ensuite, pour chacune des contraintes, le logiciel va tester la validité de son expression. Il va donc par exemple contrôler que les symboles spécifiés sont bien exacts et qu'ils sont utilisés au bon endroit, ... De façon similaire aux attributs, toute contrainte contrôlée correcte est rattachée à la feature lui correspondant dans la table des features. A la fin de cette dernière étape, l'analyse sémantique est terminée (l'objet "TVLParser" en est directement averti, flèches vertes dans les Figures 6.6 et 6.7). Les outputs sont donc les trois tables des symboles correspondant au modèle contenu dans le fichier TVL. Ces tables peuvent alors être utilisées pour générer la forme normale du modèle, comme expliqué dans le chapitre suivant.

Chapitre 7

Génération de la forme normalisée d'un modèle

Le présent chapitre va permettre d'introduire la seconde fonctionnalité du parseur TVL : la génération de la forme normalisée d'un modèle. Une telle forme présente l'avantage de pouvoir être soumise, moyennant quelques modifications, à un solveur. Dans le cas du parseur TVL, cette forme est employée afin de soumettre le modèle à un solveur de type SAT (voir chapitre suivant). Dans ce chapitre, la première section expliquera le principe de la forme normale tandis que la seconde exposera de façon globale le fonctionnement de la normalisation au sein du parseur TVL.

7.1 Principes de la forme normale

Les modèles en forme normalisée sont des modèles TVL construits à l'aide d'un sous-ensemble des constructeurs de TVL. La notation de ces modèles est très fine, c'est une syntaxe "épurée" qui n'est pas sans rappeler une notation mathématique. Par exemple, les modèles TVL non-normalisés peuvent être modélisés en utilisant des constantes et des types. A l'inverse, dans un modèle TVL normalisé, il n'est plus possible d'utiliser ce genre de mécanismes de modélisation.

Comme expliqué dans la Sous-section 4.2.1, un langage représente l'ensemble des mots qu'une grammaire est capable de générer. Dans le cas de TVL, un mot équivaut à un modèle. Ainsi, il est possible de définir le langage L_{TVL} comme l'ensemble des modèles en forme normale de TVL [8].

Définition 7.1 *Chaque modèle en forme normale est formellement défini [8] à l'aide d'un tuple* $(N, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ *où :*

- N *est l'ensemble (non vide) des features.*
- $r \in N$ *est la feature racine.*
- $DE \subseteq N \times N$ *représente les relations de hiérarchie entre les features. Par exemple, le couple* $(n, n') \in DE$ *indique que* n *est la feature parente de* n'*. Une telle relation peut aussi s'écrire* $n \to n'$*.*
- $\omega : N \to \{0, 1\}$ *marque les features optionnelles avec 1.*

- $\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$ *représente l'opérateur de décomposition d'une feature. Pour une feature donnée, il renvoie la cardinalité de cette dernière* $\langle i, j \rangle$ *où* i *représente le nombre minimum de features enfants qu'il est possible de sélectionner tandis que* j *représente le nombre maximum.*
- A *est l'ensemble des attributs.*
- $\rho : A \rightarrow N$ *est une fonction totale qui renvoie la feature à laquelle un attribut est attaché.*
- $\tau : A \rightarrow \{entier, réel, enumératif, booléen\}$ *est une fonction qui renvoie le type d'un attribut donné.*
- $\Phi \in L_{exp}$ *est une expression booléenne concernant les features de* N *et les attributs de* A*. Elle représente l'ensemble des contraintes additionnelles qui sont appliquées au modèle.*

Parallèlement, les modèles en forme normale doivent respecter un ensemble de règles :

- r *est la racine unique* $\forall n \in N(\nexists n' \in N \bullet n \rightarrow n') \Leftrightarrow n = r$.
- r *n'est pas optionnel* $\omega(r) = 0$.
- DE *ne contient pas de cycles* $\nexists n_1, ..., n_k \in N \bullet n_1 \rightarrow ... \rightarrow n_k \rightarrow n_1$.
- *Les noeuds terminaux possèdent une décomposition* $\langle 0..0 \rangle$.

L_{exp} est le langage reprenant l'ensemble des expressions booléennes valides B concernant les features de N et les attributs de A. Chacune de ces expressions doit être formée en respectant la grammaire de la Définition 7.2 où $n \in N$ est une feature, $a \in A$ est un attribut, $d \in \mathbb{Z}$ est un entier, $q \in \mathbb{Q}$ est un nombre rationnel et t est une valeur d'un attribut énumératif.

Définition 7.2 *Grammaire du langage d'expressions* L_{exp} *:*

```
B =  true | false | n | a | t | E in S | B && B | B || B | !B
   | B -> B | B <- B | B <-> B | E == E | E != E | E <= E
   | E < E | E >= E | E > E | and(B [, B]*) | or(B [, B]*)
   | xor(B [, B]*) | n excludes n | n requires n

E =  n | t | a | d | q | E + E | E - E | E / E | E * E
   | -E | abs(E) | B ? E : E | sum(E [, E]*) | mul(E [, E]*)
   | min(E [, E]*) | max(E [, E]*)

S = { E [, E]*} | [(d | *)..(d | *)] | [(q | *)..(q | *)]
```

Comme on peut le constater, un modèle normalisé ne peut être construit qu'à l'aide des constructeurs TVL permettant de définir :

- Les features et leurs relations de hiérarchie (N, r et DE).
- Le caractère optionnel des features (ω).
- Les cardinalités des features (λ).
- Les attributs de type basique (A, ρ et τ).
- Une et une seule expression booléenne (Φ) représentant l'ensemble des contraintes additionnelles portant sur N et A. Cette seule expression devra respecter la syntaxe de L_{exp} (voir Définition 7.2). Cette dernière est

moins fournie que la syntaxe originale (voir Sous-section 4.2.2) des expressions de TVL. Par exemple, il n'est pas possible d'utiliser les mots-clés "`selectedchildren`" et "`children`".

Ce qui est le plus intéressant ici, c'est que L_{TVL} est en fait un sous-ensemble du langage TVL original. En appliquant toute une liste de transformations (voir Annexe E), il est possible de générer l'équivalent en forme normale de n'importe quel modèle TVL [8]. Cette liste de transformations répertorie donc toutes les équivalences entre les constructeurs TVL interdits dans un modèle normalisé et les constructeurs TVL permis dans un modèle normalisé. Afin de pouvoir transformer n'importe quel modèle TVL en son équivalent normalisé, le parseur TVL va donc appliquer cette liste de transformations, comme expliqué dans la section suivante.

7.2 Génération de la forme normale d'un modèle par le parseur TVL

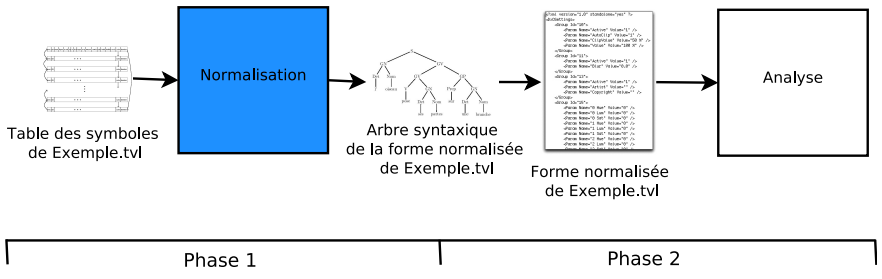

FIGURE 7.1 – Schéma illustrant le fonctionnement global de la normalisation.

Comme illustré dans la Figure 7.1, le parseur TVL génère la forme normalisée en deux étapes. Afin de faciliter la compréhension, le lecteur peut se munir de la liste des règles de transformation d'un modèle normalisé disponible dans l'annexe E.

Durant la première étape, à partir de la table des features générée à la fin de l'analyse sémantique, le parseur TVL va produire l'arbre syntaxique correspondant à la forme normalisée du modèle. Chaque "FeatureSymbol" de la table va donc être convertie en son équivalent normalisé dans l'arbre syntaxique. L'intérêt de cette démarche est que :

1. Toutes les méthodes permettant de créer un arbre syntaxique existent déjà.
2. Il est aisé de générer un fichier TVL à partir d'un arbre syntaxique.
3. La première fonctionnalité (l'analyse syntaxique et sémantique) du parseur permet de facilement contrôler le fichier TVL nouvellement généré.
4. Le fichier peut être sauvegardé et ainsi être réutilisé pour d'autres applications.
5. La table des features ne doit être que très peu modifiée. La très grande majorité des modifications apportées aux éléments (attributs, features, ...)

afin de les normaliser sont représentées dans l'arbre syntaxique et non dans la table des features.

Si les transformations concernant la normalisation étaient directement effectuées dans la table des features, à chaque fois, il faudrait d'abord effectuer une copie de la table, afin de pouvoir ultérieurement consulter les informations (correspondant au modèle non normalisé) qu'elle contient. De plus, il faudrait créer de nouvelles méthodes capables de vérifier les données qui ont été modifiées dans la table afin de normaliser le modèle. L'unique intérêt de cette approche est qu'il n'est pas nécessaire de passer par des structures de données intermédiaires (arbre syntaxique et fichier). Etant donné les avantages de la première approche (normalisation du modèle à l'aide d'un nouvel arbre syntaxique), l'auteur de ce mémoire l'a préférée à la seconde approche (normalisation directe dans la table des features du modèle non normalisé).

Comme expliqué précédemment, le parseur TVL va utiliser comme input la table des features du modèle. Plus précisément, dans cette table, il va se servir de la "FeatureSymbol" correspondant à la feature racine du modèle. Cette stratégie présente trois avantages majeurs :

1. A partir du moment où la table des features a été générée, il est certain que tous les "`include`" du fichier de base ont été résolus (voir Section 6.3). La première transformation de la liste des transformations, imposant de résoudre les "`include`", n'est donc plus à accomplir.
2. Les symboles contenus dans la table ne comportent pas d'erreurs. Durant l'analyse syntaxique et sémantique, de nombreuses vérifications ont été faites afin de s'en assurer.
3. Comme expliqué dans la Section 6.2, chaque "FeatureSymbol" de cette table contient les références vers les "FeatureSymbol" correspondant à ses enfants. De plus, chaque "FeatureSymbol" centralise tous les éléments (attributs et contraintes) spécifiés dans les différentes déclarations de la feature qu'elle représente. Donc, la "FeatureSymbol" de la feature racine du modèle "contient" (via les références de ses features enfants et les références de ses enfants vers leurs propres enfants, ...) l'ensemble des relations de hiérarchie qu'il existe entre les features. De cette manière, il ne faut plus appliquer la transformation numéro onze de la liste vu que cette dernière a déjà été "implicitement" exécutée. Cette transformation impose de fusionner les différentes déclarations d'une même feature en une seule déclaration globale. De plus, elle impose aussi de réunir ensemble les déclarations globales des features dans une seule et même structure. Dans un modèle normalisé, une feature ne peut donc être déclarée qu'une seule fois, et cela, dans le groupe de features enfants de sa feature parente.

A partir de cette table, la première chose que le parseur TVL va faire, c'est transformer les types et attributs structurés comme indiqué dans la transformation numéro trois (synthétisée dans le Tableau 7.1, situé sur la page suivante). Il est nécessaire d'accomplir cette transformation en premier car elle nécessite de modifier les attributs structurés dans la table des features. Cette opération est la seule qui nécessite d'apporter directement des modifications dans la table

des features. Si cette tâche n'est pas accomplie en premier lieu, les expressions des attributs et des contraintes risquent d'utiliser des identifiants n'existant pas ou dépréciés.

TABLE 7.1 – Normalisation d'un type et d'un attribut structurés (voir Annexe E).

Version non normalisée	
Type	Attribut

`struct t {`
`  int b;`
`  bool c;`
`}`
	`t a;`

Version normalisée	
Type	Attribut
Supprimé	`int a_b;` `bool a_c;`

Ensuite, pour chaque "FeatureSymbol" (en commençant par la feature racine), la suite des transformations accomplies afin de générer son équivalent normalisé dans l'arbre syntaxique va être la suivante [1] :

1. Dans le cas où la "FeatureSymbol" possède des enfants. Un nouveau "`FEATURE_GROUP`" va être créé afin d'accueillir :

   (a) La version normalisée de la cardinalité du groupe "`CARDINALITY`" en appliquant la transformation numéro dix (synthetisée dans le Tableau 7.2).

TABLE 7.2 – Normalisation des cardinalités.

Version non normalisée	Version normalisée
`oneof`	`[1..1]`
`someof`	`[1..*]`
`allof`	`[*..*]`

   (b) La version normalisée "`FEATURE`" de chaque "FeatureSymbol" enfant.

2. Chaque attribut de la "FeatureSymbol" va être normalisé en "`BASE_ATTRIBUTE`" en appliquant les transformations numéro deux, trois, quatre, cinq, sept et huit (partiellement synthétisées grâce à un exemple dans le Tableau 7.3, situé sur la page suivante). Chacune des contraintes créée suite à la normalisation d'un attribut sera rattachée à la contrainte globale Φ de la "`FEATURE`" racine.

1. Ici, pour plus de facilité, les noms des symboles non-terminaux de la grammaire seront utilisés à la place des noms des classes java leur correspondant. Voir package "SyntaxTree" dans la Section 5.4)

TABLE 7.3 – Normalisation d'un attribut (voir Annexe E).

Version non normalisée

`f {`
`  int a in [1..9], ifin: is 5, ifout: in [4..6];`
`}`

Version normalisée

`f {`
`  int a;`
`}`

`Φ && (c.a in [1..9]) && (c -> c.a == 5) && (!c -> c.a in [4..6])` avec `c` un chemin de features non ambiguës menant à la feature `f`.

3. Chaque contrainte de la "FeatureSymbol" va être normalisée en "`CONSTRAINT`" en appliquant les transformations numéro deux, trois, six, sept, huit et neuf (partiellement synthétisées grâce à un exemple dans le Tableau 7.4). Chacune de ces "`CONSTRAINT`" est ensuite rattachée à la contrainte globale Φ de la "`FEATURE`" racine.

TABLE 7.4 – Normalisation d'une contrainte (voir Annexe E).

Version non normalisée

`f {`
`  int a;`
`  ifin: this.a == sum(children.a);`
`  group [*..*] {f1 {int a}, f2 {int a}}`
`}`

Version normalisée

`f {`
`  int a;`
`  group [*..*] {f1 {int a}, f2 {int a}}`
`}`

`Φ && (c -> (c.a == sum(c.f1.a, c.f2.a)))` avec `c` un chemin de features non ambiguës menant à la feature `f`.

4. Une nouvelle "`FEATURE`" correspondant à la "FeatureSymbol" est créée. Ensuite, le "`FEATURE_GROUP`" (s'il y en a un) et les "`BASE_ ATTRIBUTE`" (s'il y en a) dernièrement générés sont rattachés à cette "`FEATURE`".

A la fin, lorsque la "`FEATURE`" correspondant à la "FeatureSymbol" racine a été créée et complétée de ses éléments (attributs, contrainte globale Φ et features enfants), la construction de l'arbre syntaxique est terminée et le modèle peut être considéré comme normalisé. Comme il est possible de le constater, les transformations de normalisation ne sont pas appliquées dans l'ordre dans lequel elles ont été numérotées. Si c'était le cas, il faudrait parcourir plusieurs fois la table des features. Afin de contourner ce problème, certaines transformations sont donc appliquées simultanément. Enfin, il ne pas faut oublier que grâce à la méthode de normalisation de l'interface "Expression" que toutes les classes

d'expression implémentent directement ou indirectement (voir Section 5.4), la forme normale d'une expression peut être obtenue facilement.

Après la génération de l'arbre syntaxique, la seconde étape de la normalisation commence. Cette dernière est surtout une étape de contrôle du modèle normalisé. En effet, l'arbre syntaxique est converti en un fichier TVL. Ensuite, ce fichier est soumis au parseur TVL afin d'être analysé (syntaxiquement et sémantiquement). De cette manière, il est possible de s'assurer que le modèle normalisé est bien correct. De plus, à la fin de cette analyse, la table des features correspondant au modèle normalisé est générée. Cette dernière peut alors être réutilisée pour générer la forme booléenne du modèle normalisé. Cette nouvelle forme est abordée dans le chapitre suivant.

Chapitre 8

Génération de la forme booléenne et utilisation d'un solveur SAT

Le présent chapitre va permettre de présenter en détails la troisième et dernière fonctionnalité du parseur TVL : la soumission d'un modèle à un solveur SAT. En partant de la table des features d'un modèle en forme normale, à certaines conditions, le parseur TVL peut générer la forme booléenne de ce modèle. Cette nouvelle forme peut ensuite être transformée en un problème SAT afin d'être soumise à un solveur du même nom. A ce moment-là, il devient alors possible de calculer différentes opérations par rapport au modèle. La première section de ce chapitre sera donc consacrée à la génération de la forme booléenne tandis que la seconde présentera la génération d'un problème SAT.

8.1 Génération de la forme booléenne

Cette section sera subdivisée en deux sous-sections. La première présentera le principe de la forme booléenne tandis que la seconde expliquera de façon globale comment le parseur TVL génère la forme booléenne d'un modèle normalisé.

8.1.1 Principe de la forme booléenne

La principale motivation derrière la création de la forme booléenne est la possibilité de soumettre cette dernière, après quelques transformations, à un solveur SAT. Pour rappel, à partir d'une formule booléenne φ en forme normale conjonctive, ce type de solveur est capable de calculer une ou plusieurs valuations des variables rendant la formule satisfiable. Chaque valuation représente une solution du problème SAT. S'il en existe au moins une, le problème est dit satisfiable. Une formule booléenne est sous forme normale conjonctive (FNC) si elle est une conjonction de disjonctions, c'est-à-dire : $D_1 \wedge D_2 \wedge ... \wedge D_n$ où chaque D_i est de la forme $l_1 \vee l_2 \vee ... \vee l_m$ et où les l_i sont des variables booléennes.

Dès lors, la forme booléenne d'un modèle ne peut contenir que des attributs booléens et des expressions booléennes sous forme normale conjonctive. Tous les

modèles ne sont pas compatibles, ceux possédant des attributs entiers et réels ne peuvent pas être convertis. Cette limitation provient du fait que ces attributs sont compliqués voire impossibles à convertir en booléens. Pour qu'un modèle en forme normale soit compatible, il ne peut donc contenir que des attributs booléens ou énumératifs. A l'aide d'une série de transformations, ces derniers peuvent être convertis en booléens.

Pour un attribut énumératif, chacune de ses valeurs peut être transformée en un attribut booléen et une contrainte peut être générée afin de s'assurer qu'un seul de ces attributs puisse être évalué à vrai (voir plus bas, première règle de génération d'un modèle booléen).

Pour un attribut numérique (réel ou entier), tout se complique. Une des principales difficultés provient du domaine de ces attributs. Pour un attribut :

- Entier, il faudra générer un booléen pour chacune des valeurs de son domaine (tout comme pour les valeurs des attributs énumératifs). Si le domaine de l'attribut n'est pas borné et est donc formé d'un nombre infini de valeurs, il faudra produire un nombre infini de booléens, ce qui n'est pas possible.
- Réel, même si son domaine de valeurs est borné, il faudra tout de même généré un nombre infini de booléens.

Face à un modèle contenant des attributs réels et/ou entiers, il est préférable voire obligatoire d'utiliser un solveur acceptant directement ce genre d'attributs (voir Chapitre 10).

Dans la Section 5.2, aucune exigence n'imposait de générer la forme booléenne d'un modèle. Cependant, durant le développement du parseur TVL, cette forme présentait certains avantages qui ont poussé l'auteur de ce mémoire à l'implémenter :

1. Elle permet une simplification du travail. Au moment de la génération de cette forme, seuls les attributs énumératifs, la contrainte globale Φ et les features optionnelles sont transformés en leur équivalent booléen. La structure du modèle et les cardinalités ne sont converties en leur équivalent booléen que lors de la génération du problème SAT (voir sous-section suivante).
2. Elle est intelligible par l'homme. Si tout le modèle était directement converti en un problème SAT, il serait compliqué d'examiner les transformations effectuées car le format DIMACS (voir section suivante) imposé par le solveur SAT est difficilement compréhensible.
3. Grâce à la première fonctionnalité du parseur TVL (l'analyse syntaxique et sémantique, voir Chapitre 6), il est aisé de contrôler automatiquement cette forme et d'obtenir des informations de statistiques la concernant. Si le modèle était directement converti en un problème SAT, il serait plus compliqué de vérifier les transformations et de récupérer des renseignements. Il serait par exemple nécessaire de créer de nouvelles méthodes d'analyse d'un problème SAT alors que la première fonctionnalité du parseur permet de déjà accomplir cela avec un modèle TVL.

Définition 8.1 *Formellement, tout comme pour un modèle en forme normale (voir Définition 7.1, page 89), un modèle en forme booléenne peut être défini à l'aide d'un tuple* $(N, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ *. Cependant, dans le cas d'un modèle en forme booléenne :*

- $\omega : N \rightarrow \{0\}$*. Toutes les features sont donc obligatoires.*
- $\tau : A \rightarrow \{booléen\}$*. Tous les attributs sont donc de type booléen.*
- $\Phi \in L_{expb}$

Ensuite, un modèle en forme booléenne doit aussi respecter les règles spécifiées dans la Définition 7.1. Cependant, il existe une règle spécifique aux modèles booléens. L'expression booléenne Φ *doit être en forme normale conjonctive :*

$$\Phi = C_1 \texttt{ \&\& } C_2 \texttt{ \&\& } ... \texttt{ \&\& } C_n$$

où chaque C_i *est de la forme* $x_1 \texttt{ || } x_2 \texttt{ || } ... \texttt{ || } x_m$ *et chaque* $x_j \in \{true, false, e, \texttt{!}e\}$ *où* $e \in N \cup A$*.*

Les expressions du langage L_{expb} sont formées à l'aide de la grammaire de la Définition 8.2 où $n \in N$ est une feature et $a \in A$ est un attribut.

Définition 8.2 *Grammaire du langage d'expressions* L_{expb} *:*

```
B = true | false | n | a | B && B | B || B | !B
```

Grâce à la Définition 8.1, il est possible de remarquer qu'en plus de se limiter à des attributs booléens et à une expression globale Φ sous forme normale conjonctive, la forme booléenne d'un modèle normalisé ne peut pas contenir de features optionnelles. [1].

Tout comme pour la forme normale, il existe un ensemble de règles (disponible ci-dessous) permettant de convertir un modèle en forme normale en son équivalent en forme booléenne. Comme illustré dans la Sous-section 8.1.2, c'est en se basant sur ces règles que le parseur TVL va être capable de construire la forme booléenne d'un modèle normalisé.

Règles de génération d'une forme booléenne

L'ensemble des règles ci-dessous permet de transformer un modèle normalisé en son équivalent en forme booléenne. Cependant, il ne faut pas oublier que les modèles normalisés possédant des attributs entiers ou réels ne peuvent pas être convertis. Même en appliquant ces règles, il ne sera donc pas possible de générer la forme booléenne de ce type de modèles.

1) Chaque attribut énumératif a `in` $\{v_1, ..., v_n\}$ doit être décomposé. Un attribut booléen doit être créé pour chacune des valeurs de l'attribut énumératif :
bool a_v_1 ;
...
bool a_v_n ;

1. Une autre solution aurait été d'autoriser les features optionnelles dans les modèles en formes booléennes. Elles n'auraient alors été transformées en leurs équivalents booléens (voir transformation numéro 3 page suivante) que lors de la génération du problème SAT.

De plus, il faut générer une contrainte exprimant qu'un et un seul de ces attributs booléens peut être simultanément vrai :
$(a\_v_1$ `&&` `!`a_v_2 `&&` `...` `&&` `!`$a\_v_n)$ `||`
$(a\_v_2$ `&&` `!`a_v_1 `&&` `!`a_v_3 `&&` `...` `&&` `!`$a\_v_n)$ `||`
$\vdots$
$(a\_v_n$ `&&` `!`a_v_1 `&&` `...` `&&` `!`$a\_v_{n-1})$

2) Chaque feature optionnelle `opt F` doit être remplacée par une feature "artificielle". Cette dernière doit avoir comme unique enfant la feature `F` avec une cardinalité `[0..1]`, comme illustré dans l'exemple ci-dessous.

```
...
ParentArtificielF {
    group  [0..1] {
        F
    }
}
...
```

3) Toutes les expressions booléennes dites "complexes" de la contrainte globale Φ_{NF} [2] du modèle normalisé doivent être transformées en leur équivalent simple comme indiqué dans le Tableau 8.1. Dans ces règles, les b_i sont des expressions booléennes et les n_i sont des features.

TABLE 8.1 – Forme booléenne : règles de transformation des expressions booléennes.

	Ancienne expression	Expression booléenne simple équivalente
A	b_1 `==` b_2	$b_1 \Leftrightarrow b_2$
B	b_1 `!=` b_2	`!`$(b_1 \Leftrightarrow b_2)$
C	`and`$(b_1, ..., b_n)$	$(b_1$ `&&` `...` `&&` $b_n)$
D	`or`$(b_1, ..., b_n)$	$(b_1$ `||` `...` `||` $b_n)$
E	`xor`$(b_1, ..., b_n)$	$(b_1$ `&&` `!`b_2 `&&` `...` `&&` `!`$b_n)$ `||` $(b_2$ `&&` `!`b_1 `&&` `!`b_3 `&&` `...` `&&` `!`$b_n)$ `||` $\vdots$ $(b_n$ `&&` `!`b_1 `&&` `...` `&&` `!`$b_{n-1})$
F	b_1 `?` b_2 `:` b_3	$(b_1 \Rightarrow b_2)$ `&&` $(!b_1 \Rightarrow b_3)$
G	n_1 `excludes` n_2	`!`n_1 `||` `!`n_2
H	n_1 `requires` n_2	`!`n_1 `||` n_2

Comme illustré dans le tableau ci-dessus, la version simplifiée d'une expression ne peut contenir que des opérateurs "basiques" : "`->`", "`<-`", "`<->`", "`!`", "`&&`" et "`||`". De cette manière, il est possible de directement générer l'équivalent en FNC d'une telle expression en appliquant la cinquième transformation.

2. Dans ce chapitre, la contrainte globale de la forme normalisée est notée Φ_{NF} afin de pouvoir la distinguer de la contrainte globale Φ du modèle en forme booléenne.

4) Soit les attributs énumératifs a_1 `in` $\{v_1, ..., v_n\}$, a_2 `in` $\{v_1, ..., v_n\}$. Ces attributs ont été transformés en attributs booléens conformément à la première règle de transformation (ici, par souci de concision, les contraintes ont été omises) : bool $a_1\_v_1$; ... bool $a_1\_v_n$; bool $a_2\_v_1$; ... bool $a_2\_v_n$;

Chacune des expressions de Φ_{NF} faisant référence à ces attributs doit être transformée en son équivalent booléen simple comme indiqué dans le Tableau 8.2.

TABLE 8.2 – Forme booléenne : règles de transformation des expressions utilisant des attributs énumératifs.

	Ancienne expression	Expression booléenne simple équivalente
A	a_1 `==` v_j	$a_1\_v_j$
B	a_1 `==` a_2	$(a_1\_v_1$ `&&` $a_2\_v_1)$ `||` ... `||` $(a_1\_v_n$ `&&` $a_2\_v_n)$
C	a_1 `!=` v_j	`!`$a_1\_v_j$
D	a_1 `!=` a_2	`!`$((a_1\_v_1$ `&&` $a_2\_v_1)$ `||` ... `||` $(a_1\_v_n$ `&&` $a_2\_v_n))$
E	a_1 `in` $\{v_i, v_j\}$	$a_1\_v_i$ `||` $a_1\_v_j$

5) L'expression globale Φ_{NF} doit être transformée en son équivalent en forme normale conjonctive. Pour cela, il faut successivement :

1. Supprimer les implications en les simplifiant (règles a, b et c du Tableau 8.3).
2. Distribuer les négations en appliquant les lois de Morgan (règles d et e du Tableau 8.3).
3. Distribuer les disjonctions (règles f et g du Tableau 8.3)

TABLE 8.3 – Forme booléenne : règles utilisées pour transformer une expression simplifiée en son équivalent sous forme normale conjonctive.

	Expression	Simplification
a)	$A \Rightarrow B$	`!`A `||` B
b)	$A \Leftrightarrow B$	$(!A$ `||` $B)$ `&&` $(!B$ `||` $A)$
c)	$A \Leftarrow B$	`!`B `||` A
d)	`!`$(A$ `&&` $B)$	$(!A)$ `||` $(!B)$
e)	`!`$(A$ `||` $B)$	$(!A)$ `&&` $(!B)$
f)	A `||` $(B$ `&&` $C)$	$(A$ `||` $B)$ `&&` $(A$ `||` $C)$
g)	$(B$ `&&` $C)$ `||` A	$(A$ `||` $B)$ `&&` $(A$ `||` $C)$

8.1.2 Fonctionnement global de la génération de la forme booléenne au sein du parseur TVL

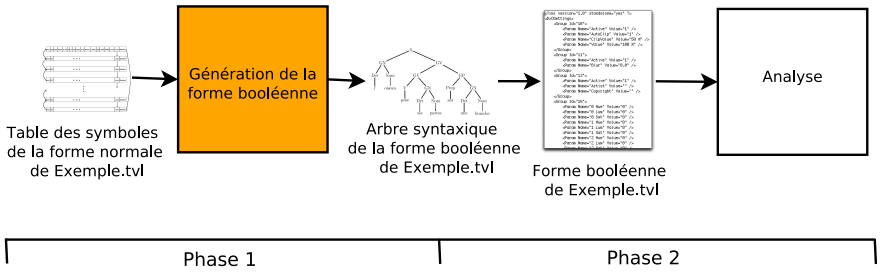

FIGURE 8.1 – Schéma illustrant le fonctionnement global de la génération de la forme booléenne d'un modèle.

Au sein du parseur TVL, la génération de la forme booléenne est fort similaire à la génération de la forme normalisée comme illustrée sur la Figure 8.1. Lors de la première phase, à partir de la table des symboles de la forme normalisée, le parseur TVL va produire l'arbre syntaxique correspondant à la forme booléenne du modèle. Ensuite, durant la seconde phase, la forme booléenne du modèle va être analysée afin de s'assurer qu'elle ne contient pas d'erreurs.

Au commencement de la phase une, la première chose que le parseur TVL va faire, c'est transformer tous les attributs énumératifs en attributs booléens comme indiqué dans la règle de transformation numéro une (voir sous-section précédente). Cette tâche doit absolument être la première à être accomplie car elle nécessite d'apporter des modifications directement dans la table des features (création de nouveaux attributs, ...). Dans le cas contraire, le modèle booléen construit risquerait de ne pas être valide. Cette opération est la seule qui nécessite d'apporter des changements dans la table des features. Toutes les autres transformations sont directement représentées dans l'arbre syntaxique de la forme booléenne et non dans la table des features.

Ensuite, chaque "FeatureSymbol" (en partant de la racine) et chaque attribut de la table des features vont être transformés en leur équivalent dans l'arbre syntaxique. Leur transformation est similaire à celle appliquée lors de la normalisation (chaque "FeatureSymbol" est transformée en une "`FEATURE`", ...) sauf que dans ce cas, toutes les transformations propres à la forme normalisée ne sont bien sûr pas appliquées. Si le parseur TVL rencontre une feature optionnelle, il applique la deuxième règle de transformation (voir sous-section précédente).

Après s'être occupé des features et des attributs, le parseur TVL va récupérer la contrainte globale Φ_{NF} de la "FeatureSymbol" racine de la table des features. A ce moment-là, le logiciel va générer la version simplifiée de l'expression Φ_{NF}. Grâce à la méthode "toSimplifiedForm()" de l'interface "BooleanExpression" que toutes les expressions compatibles avec la forme booléenne implémentent, il n'est pas trop compliqué d'obtenir la version simplifiée de Φ_{NF}. Pour pouvoir générer cette dernière, il faut appliquer les règles trois et quatre (voir sous-section précédente).

Ensuite, le parseur TVL va générer la forme normale conjonctive de l'ex-

pression simplifiée de Φ_{NF} (règle numéro cinq des transformations booléennes, voir sous-section précédente). Pour cela, il va d'abord simplifier toutes les implications (règles a, b et c du Tableau 8.3). Par après, il va distribuer les négations en appliquant la loi de Morgan [3] (règles d et e du Tableau 8.3). Enfin, il finira par distribuer les disjonctions (règles f et g du Tableau 8.3) et la génération de la forme normale conjonctive de Φ_{NF} sera alors terminée (toutes ces transformations sont effectuées via les méthodes "removeArrows()", "distributeDisjunctions()" et "distributeNegations()" de l'interface "BooleanExpression"). L'expression globale Φ nouvellement générée sera rattachée à la feature racine contenue dans l'arbre syntaxique. A ce moment-là, le parseur TVL aura terminé de générer la forme booléenne. La première phase sera donc accomplie.

Durant la seconde phase, le parseur TVL va transformer l'arbre syntaxique de la forme booléenne en un fichier TVL et ce dernier va être analysé afin de s'assurer qu'il ne contient pas d'erreurs. Si c'est le cas, le parseur TVL produira la table des features correspondant au modèle. Cette table pourra alors être réutilisée pour générer le problème SAT (voir section suivante).

8.2 Génération d'un problème SAT

A la fin de la génération de la forme booléenne, le parseur TVL va transformer le modèle booléen nouvellement généré en un problème SAT (noté φ) afin de le soumettre à un solveur du même nom. Grâce à cela, il va être possible de déterminer la satisfiabilité du modèle et de calculer diverses opérations.

Ci-dessous, la première sous-section présentera le format de formules spécifique utilisé par le solveur SAT4J [4] (un solveur SAT implémenté en Java). Ensuite, la deuxième sous-section expliquera comment générer les formules booléennes correspondant à la structure et aux cardinalités du modèle. Enfin, la troisième sous-section sera consacrée à la transformation d'un modèle booléen en un problème SAT par le parseur TVL tandis que la quatrième et dernière sous-section listera l'ensemble des opérations réalisables par le solveur SAT.

8.2.1 Format DIMACS

Le solveur SAT4J utilisé (tout comme de nombreux solveurs) n'accepte que des problèmes booléens sous format DIMACS. Ce type de format impose que :

1. L'expression booléenne représentant le problème soit en forme normale conjonctive.
2. Chaque variable du problème soit identifiée par un entier.

Afin de pouvoir soumettre le problème représentant un modèle TVL booléen au solveur SAT, il est donc nécessaire d'attribuer un identifiant numérique distinct à chaque feature et attribut (en plus de leur identifiant textuel). Par exemple, pour pouvoir soumettre la contrainte TVL (a) de la Figure 8.2 (située sur la page suivante), il a fallu doter chaque élément d'un identifiant numérique : la feature "Peugeot" possède l'identifiant "1", la feature "Renault" l'identifiant "2" et la feature "Porsche" l'identifiant "3". L'expression DIMACS qui en résulte

3. Site web concernant la loi de Morgan : `http://fr.wikipedia.org/wiki/Lois_de_De_Morgan`
4. Site web officiel de SAT4J : `http://www.sat4j.org/`

est l'expression (b) de la Figure 8.2. Chaque élément y est donc identifié par un entier. La négation d'un élément se fait à l'aide du signe "-".

```
(a) (Peugeot || Renault) && (Ferrari || !Renault)

(b) (1 || 2) && (3 || -2)
```

FIGURE 8.2 – Transformation d'une contrainte TVL en son équivalent en format DIMACS.

Au niveau de l'implémentation, chaque clause (conjonction de variables) de la formule doit être représentée dans un tableau d'entiers et tous ces tableaux peuvent être réunis dans un vecteur. Par exemple, la soumission au solveur SAT de l'expression (b) de la Figure 8.2 se fait grâce au vecteur : `{[1, 2], [3, -2]}`.

8.2.2 Transformation de la structure du graphe et des cardinalités

L'expression globale Φ du modèle booléen ne contient que des contraintes additionnelles, elle n'inclut aucune contrainte concernant la structure et les cardinalités du modèle. Dès lors, il est nécessaire de générer les formules booléennes correspondant à la structure et aux cardinalités afin de pouvoir les inclure dans le problème φ soumis au solveur SAT.

Transformation de la structure du graphe

Pour chaque feature enfant du graphe, une formule booléenne, appelée "règle de justification", va être générée. Pour une feature enfant donnée, le rôle de cette règle va être d'indiquer au solveur que s'il sélectionne la feature (i.e. il l'évalue à "vrai"), il est aussi obligé de sélectionner au moins une de ses features parentes, et ainsi de suite. Ainsi, pour la feature n dont les parents sont p_1, ..., p_n l'expression correspondant à sa règle de justification sera [29] :

$$!n \;||\; p_1 \;||\; \ldots \;||\; p_n$$

Malgré le fait que la feature racine r ne possède pas de features parentes, elle possède tout de même une règle de justification [29] :

$$r$$

Cette dernière stipule que la feature racine doit toujours être sélectionnée. Si cette règle n'existait pas, le solveur SAT pourrait toujours calculer une solution correspondant à un modèle vide. A partir de ce moment-là, n'importe quel modèle booléen posséderait donc au moins une solution, même ceux comportant des illogismes. Grâce aux règles de justification, la structure du graphe peut donc être incorporée dans le problème φ soumis au solveur. Au niveau du parseur TVL, les expressions correspondant aux règles de justification des features sont directement générées au format DIMACS (voir Sous-section 8.2.1).

Transformation des cardinalités

Parallèlement aux règles de justification, il est aussi nécessaire de générer les contraintes booléennes correspondant aux cardinalités. Autrement, le solveur pourra toujours sélectionner n'importe quelle combinaison (pouvant être vide) des features enfants d'une feature parente (ce qui correspond à une cardinalité `[0..*]`). Chaque cardinalité est encodée en une expression booléenne comme indiqué dans [29]. Le Tableau 8.4 contient les équivalences entre cardinalités et expressions booléennes pour une feature n possédant les enfants f_1, ..., f_m. LTS_{eq} et GTS_{eq} sont des algorithmes [5] permettant d'encoder de façon optimale des contraintes de cardinalités booléennes [25]. Une telle contrainte permet de contrôler le nombre de variables booléennes pouvant être simultanément vraies. Ici, $LTS_{eq}^{m,j}(f_1, ..., f_m)$ va générer la contrainte booléenne optimale permettant de s'assurer que maximum j features enfants f_k sont conjointement vraies tandis que $GTS_{eq}^{m,i}(f_1, ..., f_m)$ va produire la contrainte booléenne optimale assurant que minimum i features enfants f_k sont simultanément vraies.

TABLE 8.4 – Correspondances entre cardinalités et formules booléennes [29].

Cardinalité	Expression booléenne équivalente
`[0..m]`	Aucune expression nécessaire
`[0..0]`	$\bigwedge_{i=1..m}$ (`!`n `||` `!`f_i)
`[0..j]` $1 \leq j < m, m \geq 2$	`!`n `&&` $LTS_{eq}^{m,j}(f_1, ..., f_m)$
`[1..m]` $m \geq 1$	`!`n `||` f_1 `||` ... `||` f_m
`[m..m]` $m \geq 1$	$\bigwedge_{i=1..m}$ (`!`n `||` f_i)
`[i..m]` $0 < i < m, m \geq 2$	`!`n `&&` $GTS_{eq}^{m,i}(f_1, ..., f_m)$
`[i..j]` $1 \leq i \leq j < m, m \geq 2$	`!`n `&&` $GTS_{eq}^{m,i}(f_1, ..., f_m)$ `&&` $LTS_{eq}^{m,j}(f_1, ..., f_m)$

Au niveau de l'implémentation, les expressions booléennes correspondant aux cardinalités sont directement générées au format DIMACS (voir Sous-section 8.2.1).

8.2.3 Utilisation du solveur SAT

La génération du problème SAT correspondant à un modèle TVL booléen nécessite donc de transformer ce dernier en une formule booléenne φ sous format DIMACS. Pour cela, le parseur TVL va tout d'abord récupérer la contrainte globale Φ de la feature racine du modèle. Ensuite, il va générer l'équivalent DIMACS de cette contrainte. A ce moment-là, φ sera donc composé de l'équivalent en forme DIMACS de Φ.

Par après, le parseur TVL va parcourir la table des features de la forme booléenne. Pour chaque "FeatureSymbol" (en partant de la feature racine), il va générer la contrainte correspondant à sa règle de justification. De plus, si cette feature possède un groupe de features enfants, il va aussi produire la formule booléenne correspondant à la cardinalité de ce groupe. A chaque fois, les expressions générées sont rajoutées à φ (en respectant le format DIMACS). A la fin, lorsque le parseur TVL aura parcouru chacune des features de la table des

5. Ces algorithmes ont été utilisés comme des "blackbox" par le parseur TVL. Le sujet de ce mémoire n'étant pas l'encodage optimal de contraintes en FNC, pour plus d'informations à propos de ces algorithmes, le lecteur peut consulter le papier [25].

features, il soumettra le problème SAT φ créé au solveur SAT4J et ce dernier pourra alors commencer à calculer différentes opérations.

8.2.4 Opérations réalisables par le solveur SAT

Après avoir reçu le problème φ correspondant à un modèle booléen, le solveur SAT4J peut calculer différentes opérations :

1. **La satisfiabilité du modèle** : Le solveur va tenter de trouver au moins une combinaison de features et d'attributs permettant de satisfaire φ. S'il y arrive, le modèle est dit satisfiable. Dans ce cas, il est alors possible de calculer les opérations des points suivants. Chaque combinaison de features et d'attributs représente une solution de φ et donc du modèle. Vu qu'un modèle de features est une représentation de haut niveau d'une SPL (au niveau des caractéristiques et des exigences), chaque solution représente donc un produit développable par la SPL.
2. **Le nombre de solutions** : Le solveur renvoie le nombre de combinaisons de features et d'attributs permettant de satisfaire φ, c'est-à-dire, le nombre de produits que la SPL est capable de développer. De façon plus précise, il est aussi possible d'obtenir le nombre de solutions incluant une feature déterminée.
3. **La génération des solutions** : Le solveur génère chaque combinaison de features et d'attributs permettant de satisfaire φ. Autrement dit, il génère la liste des produits développables par la SPL. De façon plus spécifique, il est aussi possible de ne sélectionner que les solutions contenant une certaine feature.
4. **La détection des features mortes** : Une feature morte est une feature qui n'est sélectionnée dans aucune des solutions générées. A partir de l'identifiant d'une feature, le solveur peut détecter si cette dernière est morte.

Troisième partie

Conclusion

Chapitre 9

Evaluation et proposition d'extension du langage TVL

Après la présentation du parseur TVL et de chacune de ses fonctionnalités, ce premier chapitre de la partie de conclusion de ce mémoire va permettre de discuter du langage de modélisation utilisé : TVL. Ci-dessous, la première section permettra d'évaluer TVL et d'analyser ses limites. Ensuite, dans la seconde section, une proposition d'extension de TVL sera faite afin de rendre possible la modélisation des différents aspects de la variabilité introduits dans le Chapitre 3.

9.1 Evaluation de TVL

Dans le Chapitre 4, TVL a été présenté comme possédant de nombreux avantages. Cependant, tout comme les diagrammes de features, TVL possède des limites le rendant parfois moins adapté à certains acteurs ou situations.

Syntaxe moins simple à appréhender comparée à celle des diagrammes de features

La syntaxe utilisée par TVL, proche de celle du C, nécessite d'avoir un minimum de connaissances en programmation. De plus, afin de pouvoir comprendre et manipuler un modèle TVL, il est d'abord nécessaire d'étudier la grammaire du langage et toutes les constructions que cette dernière permet. Par exemple, si un acteur n'est pas au courant qu'une même feature "`X`" peut être déclarée un nombre quelconque de fois, il pourra mal interpréter le modèle et penser qu'il existe plusieurs features "`X`". Si le modèle avait été représenté graphiquement, à l'aide des diagrammes de features, l'acteur n'aurait sans doute pas rencontré ce problème. En effet, la feature "`X`" n'aurait été représentée qu'à un seul endroit, dans une "boîte" bien délimitée. Pour une personne ayant peu de connaissances en informatique, comprendre et manipuler un modèle TVL peut donc demander des efforts. Notamment plus d'efforts que dans le cas des diagrammes où la syntaxe simplifiée (voir Chapitre 3) ne requiert normalement aucun apprentissage.

Limitation de l'aspect textuel

De plus, au-delà de la syntaxe, l'aspect textuel de TVL peut lui-même constituer un inconvénient. Les langages de modélisation graphique possèdent certains avantages pouvant faciliter la compréhension [19]. Pour un acteur ayant peu de connaissances en informatique, il sera sans doute plus facile de manipuler les différents concepts d'un modèle s'ils sont représentés graphiquement. Par exemple, dans un diagramme de features, le fait que les relations entre une feature parente et ses features enfants soient représentées graphiquement à l'aide d'arcs permet de bien matérialiser la hiérarchie des features. Même une personne n'ayant jamais utilisé ce genre de diagrammes pourra sans doute directement comprendre qu'il existe un lien de filiation entre ces features.

Dès lors, tout comme les auteurs de TVL le recommandent, dans le cadre d'un dialogue avec une personne ayant peu de connaissances en informatique, il vaut mieux utiliser les diagrammes des features. La compréhension et la communication n'en seront que plus efficaces. Cependant, entre des personnes possédant des compétences techniques, il est plus que conseillé d'employer les modèles TVL. Cela, en raison de tous les avantages cités dans la Section 4.3.

9.2 Proposition d'extension de TVL

TVL est un langage permettant de modéliser de nombreux éléments : attributs, contraintes additionnelles, types, ... Mais une question se pose, lors du domain engineering et de l'application engineering, est-il possible de modéliser les différents aspects (binding time, ...) de la variabilité présentés dans le Chapitre 3 ? Cette seconde section va permettre de répondre à cette question. Ci-dessous, la première sous-section exposera les deux approches de modélisation choisies tandis que la seconde tentera de les comparer et de désigner laquelle est la plus adaptée.

9.2.1 Modélisation des différents aspects de la variabilité

Ci-dessous, chaque sous-section traite d'un des aspects de la variabilité (voir première section du Chapitre 3) et du moyen de le modéliser à l'aide de TVL.

Modélisation de la variabilité à différents niveaux d'abstraction

Pour les ingénieurs, il peut être pratique de pouvoir indiquer à quel niveau d'abstraction se situe une feature. Par exemple, grâce à un logiciel, il pourra alors être possible de ne visualiser que les features se trouvant à un certain niveau d'abstraction ou au contraire, de les masquer. Actuellement, grâce aux blocs de données, il est déjà possible d'indiquer le niveau d'abstraction d'une feature. Par exemple, la feature "UseCaseConnexion" de l'exemple ci-dessous se trouve au niveau d'abstraction des exigences.

```
UseCaseConnexion {
    data { "abstractionLvl" "exigences"; }
}
```

Toutefois, il pourrait aussi être intéressant d'introduire une syntaxe officielle permettant de représenter les niveaux d'abstraction des features. De cette manière, les blocs de données resteraient uniquement destinés à l'usage de programmes spécifiques. Afin de ne pas complexifier les modèles et d'englober les concepts discutés dans les autres sous-sections, un bloc spécifique peut être utilisé. Ainsi, dans la feature "UseCaseConnexion" ci-dessous, le niveau d'abstraction a été indiqué dans le bloc "`properties`".

```
UseCaseConnexion {
    properties {
        abstractionLvl is exigences;
    }
}
```

L'utilisation du bloc "`properties`" présente donc l'avantage de réunir en un seul "endroit" tous les constructeurs ("`abstractionLvl`", voir sous-sections suivantes pour les autres) concernant les propriétés de variabilité d'une feature. Sans ce bloc, ces constructeurs pourraient être disséminés à travers les différentes déclarations d'une même feature, ce qui compliquerait la compréhension et l'analyse des modèles.

Modélisation de la variabilité interne/externe

Afin de discuter des exigences du programme avec le client, un développeur préférera sans doute générer le diagramme correspondant au modèle de features (ou à un sous-ensemble des features de ce modèle). Dès lors, afin de ne présenter au client que les features qui ont du sens à ses yeux, il est important de pouvoir indiquer la visibilité interne ou externe des features. De cette manière, le diagramme généré ne sera par exemple composé que des features possédant une visibilité externe. Ici, la modélisation peut aussi se faire de deux manières différentes, via un bloc de données ou via le bloc spécifique "`properties`". Dans le tableau ci-dessous, la feature "UseCaseConnexion" possède une visibilité externe.

Approche bloc de données	Approche bloc "`properties`"

`UseCaseConnexion {`
`  data {`
`  "external" "true";`
`  }`
`}`
	`UseCaseConnexion {`
	`  properties {`
	`  external is true;`
	`  }`
	`}`

Modélisation des mécanismes de variation

Lors de la modélisation d'une feature parente, un ingénieur pourra vouloir indiquer le ou les mécanisme(s) de variation permettant d'incarner "physiquement" la relation entre la feature parente et ses features enfants. Encore une fois, la modélisation des mécanismes peut se faire soit via les blocs de données, soit via le bloc spécifique "`properties`". Dans le tableau situé sur le haut de la page suivante, le mécanisme de variation de la feature parente "driverAntenne" est basé sur l'héritage des classes.

Approche bloc de données	Approche bloc "properties"

driverAntenne {
group oneof {B200, B300}
data {
"mechanism" "héritage";
}
}
	driverAntenne {
	group oneof {B200, B300}
	properties {
	mechanism is héritage;
	}
	}

Modélisation du binding time

Enfin, afin de s'assurer de la cohérence des décisions, il peut être nécessaire d'indiquer les binding times des features parentes possédant des features enfants. Ils serviront donc à signaler aux acteurs (ingénieurs, clients, utilisateurs, ...) à quel(s) moment(s) ils devront sélectionner la ou les feature(s) enfant(s) de la feature parente. La stratégie de modélisation est de nouveau basée sur deux approches : l'utilisation des blocs de données ou du bloc spécifique "properties". Dans l'exemple du tableau ci-dessous, les features enfants de la feature "Langue" doivent être sélectionnées au moment de l'installation ou de l'exécution du programme.

Approche bloc de données	Approche bloc "properties"

Langue {
group oneof {FR, EN}
data {
"bindingT" "install";
"bindingT" "exéc";
}
}
	Langue {
	group oneof {FR, EN}
	properties {
	bindingT in {install, exéc};
	}
	}

9.2.2 Comparaison des deux approches de modélisation

Dans la sous-section précédente, l'auteur de ce mémoire a proposé de modéliser chaque aspect de la variabilité de deux manières différentes, soit en utilisant les blocs de données, soit en utilisant le bloc spécifique "properties".

La première approche présente l'avantage majeur de ne nécessiter aucune modification de la grammaire de TVL. Chaque aspect de la variabilité peut être modélisé à l'aide d'un ou plusieurs couples de données (comme illustré dans la sous-section précédente). Cependant, cette approche est discutable. En effet, les blocs de données ont été prévus pour contenir des informations externes (utilisées par exemple par des programmes de visualisation, ...) et non pas des informations ayant un rapport direct avec le modèle. De plus, les informations contenues dans les blocs de données ne sont pas exploitées par le parseur TVL. Lors de l'analyse syntaxique, le parseur se contente uniquement de contrôler que chaque couple de données est bien constitué de deux chaînes de caractères. Si un des couples indique qu'une feature possède une visibilité interne, le parseur ne pourra pas exploiter cette information afin de générer un modèle dans lequel cette feature n'est pas reprise.

La deuxième approche nécessite de modifier la grammaire de TVL afin d'y ajouter le bloc "properties" et les constructeurs ("bindingT", ...) des différents aspects de la variabilité (la syntaxe nécessaire à cette modification est disponible

dans la Figure 9.1). Toutefois, cette approche présente deux avantages. Premièrement, les informations externes et celles concernant la variabilité ne sont pas mélangées dans les blocs de données. Le bloc "`properties`" permet de regrouper toutes les données correspondant aux différents aspects de la variabilité d'une feature. La représentation est beaucoup plus "propre" et la compréhension n'en est donc que facilitée. Deuxièmement, à condition de modifier le parseur TVL, ce dernier peut exploiter les informations contenues dans le bloc "`properties`". Dans ce cas, le logiciel pourra par exemple contrôler les informations incluses dans ce bloc ou générer des modèles ne reprenant que les features possédant une visibilité externe.

```
FEATURE_BODY_ITEM =  ...
                   | PROPERTIES

PROPERTIES = "properties" "{" PROPERTIES_ITEM+ "}"

PROPERTIES_ITEM =  ABSTRACTIONLEVEL
                 | VISIBILITY
                 | MECHANISM
                 | BINDINGTIME

ABSTRACTIONLEVEL =  "abstractionLvl" "is" ID ";"
                  | "abstractionLvl" "in" SET_EXPRESSION ";"

VISIBILITY = "external" "is" ("true" | "false") ";"

MECHANISM =  "mechanism" "is" ID ";"
           | "mechanism" "in" SET_EXPRESSION ";"

BINDINGTIME =  "bindingT" "is" ID ";"
             | "bindingT" "in" SET_EXPRESSION ";"
```

FIGURE 9.1 – Syntaxe TVL permettant de modéliser les différents aspects de variabilité.

Dans la grammaire de la Figure 9.1, la première règle étend TVL en permettant d'inclure le bloc "`properties`" dans le corps d'une feature (via le non-terminal "`FEATURE_BODY_ITEM`", voir grammaire de TVL dans l'annexe B). Ensuite, les deux règles suivantes ("`PROPERTIES`" et "`PROPERTIES_ITEM`") permettent de modéliser le bloc proprement dit. Par après, les règles "`ABSTRACTIONLEVEL`", "`VISIBILITY`", "`MECHANISM`" et "`BINDINGTIME`" permettent chacune de modéliser un des aspects de la variabilité comme illustré dans la sous-section précédente. Il faut noter que grâce à ces règles, pour chaque feature, un développeur peut spécifier un (via le mot-clé "`is`") ou plusieurs (via le mot-clé "in") binding time(s)/niveau(x) d'abstraction/mécanisme(s) de variation. Au niveau du parseur TVL, suite à l'introduction de cette nouvelle syntaxe, il faudra par exemple vérifier qu'il n'existe qu'un seul bloc "`properties`" dans l'ensemble des déclarations d'une même feature. De même, il faudra aussi contrôler que chaque aspect de la variabilité ("`ABSTRACTIONLEVEL`", "`VISIBILITY`", ...) n'est représenté maximum qu'une seule fois dans un bloc "`properties`".

En raison des avantages de la seconde approche, malgré les modifications nécessaires, l'auteur de ce mémoire conseille de l'adopter. Cependant, avant de rendre cette extension "officielle", il est nécessaire de s'assurer de son utilité. Dans ce but, il faut donc la proposer à des entreprises et reccueillir leurs avis quant à son intérêt. Introduire inutilement une extension risquerait de faire dévier TVL de son objectif principal : proposer une solution de modélisation en adéquation avec les besoins du monde industriel.

Chapitre 10

Evaluations et perspectives du parseur TVL

Avant le chapitre de conclusion de ce mémoire, le présent chapitre va permettre de jeter un regard critique par rapport à la solution proposée : le parseur TVL. Comme toute solution, le logiciel possède des limites qu'il est nécessaire de mettre en avant afin de pouvoir les résoudre. De cette manière, le parseur pourra être perfectionné ce qui le rendra encore plus efficace dans sa réponse à la problématique. Ci-dessous, chaque section exposera une fonction ou une propriété du parseur qu'il est possible d'améliorer et dans certains cas, une ou plusieurs pistes de solutions seront présentées.

Génération des formules

Lors de la génération de la forme booléenne, l'accent n'a pas été mis sur l'optimisation des formules en FNC. Actuellement, le parseur TVL génère l'équivalent en forme normale conjonctive de la contrainte globale Φ_{NF} d'un modèle normalisé de manière fort simpliste (voir Sous-section 8.1.2). Il commence par supprimer toutes les implications. Ensuite, il distribue les négations et finit par aussi distribuer les disjonctions. Cette transformation en plusieurs étapes garantit de générer une formule en FNC équivalente. Cependant, cette méthode présente deux défauts majeurs :

- Elle génère des formules CNF pouvant posséder une taille exponentielle par rapport à la formule originale. Par exemple, pour une formule du type :

  $$\Phi_{NF} = (f_1 \wedge g_1) \vee (f_2 \wedge g_2) \vee ... \vee (f_n \wedge g_n)$$

  Le parseur TVL générera son équivalent en forme CNF :

  $$(f_1 \vee ... \vee f_{n-1} \vee f_n) \wedge (f_1 \vee ... \vee f_{n-1} \vee g_n) \wedge ... \wedge (g_1 \vee ... \vee g_{n-1} \vee g_n)$$

  Dans cet exemple, la formule originale est composée de n conjonctions de deux variables tandis que son équivalent en FNC compte 2^n disjonctions (clauses).
- La résolution de telles formules (composées d'un nombre exponentiel de clauses) par le solveur SAT peut être lourde et nécessiter de nombreuses ressources.

Dés lors, dans le futur, il est important de trouver une méthode permettant de générer des formules optimales en FNC. De cette manière, la résolution d'une formule nécessitera moins de temps et de ressources.

Utilisation d'autres types de solveurs

Actuellement, le seul solveur utilisé est un solveur SAT et ce dernier ne peut analyser que des modèles en formes booléennes. Dès lors, comme cela a été expliqué dans la Sous-section 8.1.1, les modèles intégrant des attributs entiers ou réels ne peuvent pas être analysés. Il pourrait donc être intéressant d'utiliser d'autres types de solveurs permettant de prendre en compte ces types d'attributs. Par exemple, il serait possible d'utiliser un solveur CSP ("Constraint Satisfaction Problem"). Les problèmes CSP sont modélisés comme un ensemble de contraintes faisant intervenir différentes variables. Le rôle du solveur est donc de déterminer s'il existe au moins une valuation de ces variables respectant l'ensemble des contraintes du problème. Dans les problèmes CSP, le gros avantage, c'est qu'il est possible d'employer aussi bien des attributs booléens ou énumératifs que des attributs réels ou entiers.

Modifications en temps réel

Lorsque le parseur TVL analyse un modèle, il le fait de façon "statique", c'est-à-dire que tout le modèle doit d'abord être analysé, transformé en un problème SAT pour ensuite être soumis au solveur. Si l'utilisateur désire ajouter des éléments au modèle (features, attributs, ...), il sera obligé de réanalyser tout le modèle et de nouveau le transformer en un problème SAT pour enfin le soumettre au solveur. Cette procédure est assez lourde et contraignante. Il pourrait donc être intéressant de développer une fonctionnalité permettant de prendre en compte en temps réel les modifications apportées au modèle. Par exemple, si l'utilisateur rajoute une feature enfant à une feature parente, il faudrait directement contrôler que cet ajout ne provoque pas d'erreurs dans le modèle : il faudrait vérifier que la cardinalité du groupe est toujours respectée, que la nouvelle feature ne viole aucune contrainte, ...

Tester le parseur TVL

Une façon saine de s'assurer de la fiabilité d'un programme, c'est de le soumettre à des cas réels ou à des tests créés par des personnes extérieures à l'équipe de développement. Actuellement, le parseur TVL n'a été soumis qu'à des tests développés en interne. Le grand défaut de ce genre de test, c'est qu'il ne reflète pas la diversité des cas réels. En effet, les développeurs se concentrent parfois involontairement sur certains aspects, ou plus simplement, il n'est pas toujours possible de penser à tester tous les cas limites. Afin de s'assurer de la bonne tenue du parseur TVL, il pourrait être intéressant de le soumettre à des cas réels. Par exemple, en utilisant des modèles de features développés dans des entreprises.

Eviter les modifications des tables des features

Lors de la génération de la forme normalisée, afin de transformer un attribut structuré en son équivalent normalisé, il est nécessaire d'apporter certaines

modifications dans la table des features du modèle original (Voir Sous-section 7.2). De manière similaire, lors de la génération de la forme booléenne, afin de transformer les attributs énumératifs en attributs booléens, il est nécessaire de modifier la table des features du modèle normalisé (voir Sous-section 8.1.2). Dès lors, ces tables pourront parfois contenir des informations ne correspondant pas au modèle original ou au modèle normalisé. Afin de garder les informations de ces tables intactes, il serait intéressant de trouver un autre moyen de générer les équivalents normalisés/booléens des attributs structurés/énumératifs.

Encodage booléen des attributs énumératifs

Lors de la génération de la forme booléenne d'un modèle, pour chacune des valeurs d'un attribut énumératif, un attribut booléen est généré (voir Sous-section 8.1.1). Cependant, il est possible d'utiliser un encodage réduisant le nombre d'attributs booléens nécessaires à chaque transformation. Pour cela, il suffit de s'inspirer de l'encodage binaire des entiers pratiqué au sein d'un ordinateur. Dans cet encodage, à l'aide de n bits, il est possible de représenter tous les entiers se situant en 0 et $2^n - 1$.

Dès lors, pour représenter un attribut énumératif dont le domaine compte 2^n valeurs, il ne faudra plus que n attributs booléens. Dans le tableau ci-dessous, les valeurs de l'attribut énumératif a `in` $\{v_1, ..., v_8\}$ sont représentées à l'aide de trois variables booléennes `a, b` et `c`.

Valeur	Encodage actuel	Encodage optimisé
v_1	`bool` a_v_1	`!a && !b && !c`
v_2	`bool` a_v_2	`!a && !b &&  c`
v_3	`bool` a_v_3	`!a &&  b && !c`
v_4	`bool` a_v_4	`!a &&  b &&  c`
v_5	`bool` a_v_5	`a && !b && !c`
v_6	`bool` a_v_6	`a && !b &&  c`
v_7	`bool` a_v_7	`a &&  b && !c`
v_8	`bool` a_v_8	`a &&  b &&  c`

Dans les contraintes des règles numéro une et quatre de la Sous-section 8.1.1, chaque attribut booléen (a_v_1, $a\_v_2, ...$) correspondant à une valeur devra être remplacé par l'encodage optimisé correspondant à la valeur. Par exemple, pour la contrainte :

$$a \texttt{ == } v_1$$

L'équivalent en forme simplifiée utilisant l'encodage optimisé sera :

```
!a && !b && !c
```

Avant d'adopter ce nouvel encodage, il est important d'étudier son impact sur la résolution des problèmes par le solveur SAT. En effet, la diminution du nombre de variables est compensé par une augmentation du nombre de conjonctions (l'introduction des encodages optimisés dans les contraintes) et la transformation de ces dernières sous FNC peut entraîner à son tour un accroissement du nombre de clauses. Dans ce contexte, il est donc important de déterminer si les performances de résolution du solveur induites par la dimunition du nombre de variables ne sont pas contre balancées par l'augmentation du nombre de clauses.

Chapitre 11

Conclusion

Dans la première partie de ce mémoire, le contexte de la problématique a tout d'abord été introduit. Le paradigme des SPLs a été présenté dans le Chapitre 2. Les principes-clés (réutilisation, plateforme, variabilité, ...) et activités-clés (domain engineering, software engineering, ...) des SPLs ont été discutés. Ensuite, les différents avantages et risques liés à leur utilisation ont été exposés. Ce chapitre a aussi permis de démontrer en quoi il était intéressant de mener des recherches dans le domaine des SPLs.

Par la suite, le Chapitre 3 a été consacré à l'un des concepts-clés des lignes de produits : la variabilité. Premièrement, les différentes aspect (visibilité interne ou externe, binding time,...) de la variabilité ont été décrits. Deuxièmement, des diagrammes permettant de la gérer et de la modéliser à un haut niveau d'abstraction ont été présentés : les diagrammes de features.

Par après, dans le Chapitre 4, les défauts majeurs des diagrammes de features ont été détaillés. Ces derniers possèdent en effet certains inconvénients (limites graphiques, besoin d'outils dédiés, ...) les empêchant quelquefois d'être en adéquation avec les besoins du monde industriel. Ensuite, toujours dans le Chapitre 4, TVL, un langage textuel de modélisation basé sur les features développé par l'équipe LIEL a été présenté. Le but de ce langage est de combler les lacunes des diagrammes de features et de proposer au monde industriel une solution réellement en adéquation avec ses besoins.

Toutefois, présenté ainsi, TVL n'est pas pleinement exploitable par les entreprises. Créer ou modifier un modèle TVL ne pose aucun problème. Cependant, au moment de contrôler ce modèle, les choses se compliquent. En effet, pour un être humain, contrôler manuellement (sans le support d'un logiciel) un modèle composé de centaines d'éléments (features, attributs, ...) peut se révéler être une tâche complexe fort gourmande en ressources. De même, calculer manuellement diverses opérations (notamment la satisfiabilité) par rapport à un tel modèle peut être impossible. Dans ce contexte, les avantages de TVL risqueraient d'être voilés par les inconvénients liés au contrôle des modèles. Sans un logiciel automatisant le contrôle et l'analyse, les industriels pourraient ne trouver aucun intérêt à utiliser TVL.

Dès lors, dans la seconde partie de ce mémoire, **un logiciel a été proposé afin d'automatiser le contrôle et l'analyse de modèles TVL**. Actuelle-

ment, les **trois fonctionnalités majeures** de ce logiciel sont :

1) L'analyse syntaxique et sémantique d'un modèle :
Le parseur TVL est capable de vérifier aussi bien la forme que le contenu d'un modèle TVL (Chapitre 6). Il peut par exemple vérifier que le modèle ne contient pas de cycles ou que toutes les contraintes sont bien de type booléen. Grâce à cette fonctionnalité, le contrôle d'un modèle peut maintenant se faire automatiquement.

2) La génération de la forme normale d'un modèle :
En appliquant une liste de transformations, le parseur TVL est capable de générer la forme normalisée d'un modèle (Chapitre 7). Un tel modèle est construit en employant un sous-ensemble des constructeurs de TVL. Son principal avantage est de pouvoir être réutilisé afin d'être soumis, moyennant quelques modifications, à un solveur. Cette seconde fonctionnalité permet d'alléger le travail des développeurs en automatisant la génération de la forme normalisée.

3) Calcul de diverses opérations par rapport à un modèle :
A certaines conditions, le parseur TVL peut récupérer la forme normalisée d'un modèle afin de la soumettre, après quelques transformations, à un solveur SAT (Chapitre 8). A ce moment-là, il devient possible de calculer diverses opérations par rapport au modèle. Le parseur TVL peut notamment déterminer la satisfiabilité d'un modèle. Cette troisième fonctionnalité permet aux développeurs d'obtenir de précieuses informations d'analyse concernant un modèle.

De plus, parallèlement à ces fonctionnalités, le logiciel possède aussi **deux caractéristiques importantes** :

- **Il peut être utilisé comme une librairie**. Dès lors, la plupart des programmes pourront interagir avec lui. Par exemple, dans une société, les développeurs pourront continuer à utiliser leurs propres logiciels tout en bénéficiant des fonctionnalités du parseur TVL.
- **Il est portable**, autrement dit, il peut être utilisé sous la plupart des systèmes d'exploitation actuels.

De par ses fonctionnalités, le parseur TVL permet de réellement supporter les développeurs en automatisant le contrôle et l'analyse de modèles TVL. De plus, le fait qu'il soit portable et qu'il puisse être utilisé comme une librairie garantit qu'il s'adaptera au mieux dans l'environnement de chaque entreprise. En outre, en développant le parseur TVL, l'auteur de ce mémoire espère aussi avoir pu donner un aperçu "pratique" des avantages de TVL. Dès lors, l'auteur a bon espoir que ce logiciel puisse être réutilisé ou du moins, qu'il puisse servir de piste pour d'autres outils d'analyse de modèles TVL.

Enfin, parallèlement au parseur TVL, une **extension au langage TVL** a été présentée dans le Chapitre 9. Cette dernière propose de compléter TVL en intoduisant une syntaxe **permettant de modéliser les différents aspects de la variabilité** présentés dans le Chapitre 3.

Glossaire

Artéfact : Un artéfact peut être une architecture, un cas de test, un composant logiciel,... les artéfacts sont utilisés pour développer les produits d'une SPL. Un artéfact doit être assez flexible que pour être réutilisable dans plusieurs logiciels.

Binding time : Le binding time d'un point de variation est la période du développement durant laquelle les acteurs doivent fixer ce point de variation. Par exemple, un point de variation peut être fixé durant l'exécution de logiciel.

Contraintes additionnelles : Une contrainte additionnelle est une contrainte qui n'est pas exprimée en utilisant les cardinalités et/ou la structure du graphe. Par exemple, les contraintes traditionnelles "`excludes`" ou "`requires`" sont des contraintes additionnelles.

Domain et application engineering : Le domain engineering reprend l'ensemble des activités concernant la création et la maintenance de la SPL. Il comprend notamment les activités de développement et de maintenance des artéfacts. L'application engineering englobe l'ensemble des activités concernant la création des produits de la SPL. Il inclut donc les activités de "fixation" des points de variation, d'intégration des artéfacts, ...

Feature : D'après la définition donnée dans le rapport technique de FODA [16], une feature est : "*the attributes of a system that directly affect end-users*" ou encore "*A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems*".

Feature ambiguë : Dans un modèle TVL, une feature ou un identifiant est dit ambigu(ë) lorsqu'il existe plusieurs features possédant cet identifiant.

Feature morte : Une feature morte est une feature qui ne peut être sélectionnée dans aucun des produits de la SPL. Souvent, ceci est la conséquence de contraintes mal formées.

Ingénieurs/développeurs domaine et software : Les ingénieurs/développeurs domaine sont les professionnels employés par le domain engineering tandis que les ingénieurs/développeurs software sont les professionnels employés par l'application engineering.

Mécanisme de variation : Un mécanisme de variation représente l'implémentation "physique" d'un point de variation. En fonction de l'artéfact et de son

niveau d'abstraction, différents mécanismes de variation peuvent être utilisés pour incarner un point de variation.

Plate-forme : Une plate-forme est constituée de l'ensemble des artéfact nécessaire aux développements des produits d'une SPL. Un artéfact ne sera contenu dans la plate-forme que s'il peut être réutilisé dans plusieurs produits.

Point de variation et variant : Un point de variation est un mécanisme générique permettant de représenter la variabilité à l'intérieur d'un artéfact. Un tel point représente un choix que les acteurs (clients, ingénieurs software, ...) doivent effectuer. Les différentes possibilités de ce choix sont incarnées par les variants du point de variation. Un point de variation est dit fixé quand un de ses variants a été sélectionné. Au niveau des modèles de features, un point de variation est souvent représenté par une feature parente tandis que ses variants sont modélisés par les features enfants de cette feature parente.

Satisfiabilité d'un modèle : Un modèle de features est dit satisfiable s'il existe au moins une combinaison de features et d'attributs respectant l'ensemble des contraintes du modèle (structure, cardinalités et contraintes additionnelles). Cette combinaison correspond à un produit de la SPL représentée par le modèle de features.

Variabilité dans le temps et dans l'espace : La variabilité dans le temps d'un artéfact représente l'évolution de l'artéfact, c'est-à-dire, les différentes "versions" de l'artéfact qu'il a pu exister au cours de la vie de la SPL. La variabilité dans l'espace d'un artéfact représente les différentes formes que l'artéfact peut prendre à un moment précis. Ici, une "forme" représente un artéfact où les points de variation ont été fixés grâce à la sélection de certains variants.

Variabilité interne ou externe : La variabilité interne est la variabilité visible aux yeux du client. A l'inverse, la variabilité externe est la variabilité dont le client n'a pas conscience car elle a été volontairement masquée. Ceci peut être le cas quand la variabilité concerne des choix technologiques qui ne présentent aucun intérêt pour le client.

Bibliographie

[1] Henry ford, l'homme de la ford t. `http://www.caradisiac.com/php/voitures_collection/portraits/geants/henry_ford.php` [Online ; consulté le 09-Avril-2010].

[2] Binding time in software product lines, 2010. `http://www.softwareproductlines.com/introduction/binding.html` [Online ; consulté le 02-Juin-2010].

[3] T. Asikainen. *Modeling Methods for Managing Variability of Configurable Software Product Families*. PhD thesis, Licentiate Thesis of Science in Technology at Helsinki University of Technology, 2004.

[4] F. Bachmann and P. Clements. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Pittsburgh, USA, 2005.

[5] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use case description of requirements for product lines. In *Proceedings of the International Workshop on Requirements Engineering for Product Lines 2002 - REPL 02. Technical Report ALR2002-033, AVAYA*, pages 12–18, 2002.

[6] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 159–162. University of Duisburg-Essen, January 2010. Acceptance rate : 54.

[7] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines : a systematic review. In *SPLC '09 : Proceedings of the 13th International Software Product Line Conference*, pages 81–90, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[8] Andreas Classen, Quentin Boucher, Paul Faber, and Patrick Heymans. Syntax and semantics of TVL, a text-based feature modelling language. Technical report, PReCISE Research Center, University of Namur, Namur, Belgium, 2010. www.info.fundp.ac.be/∼acs/tvl.

[9] Matthias Clauss. Generic modeling using uml extensions for variability. In *OOPSLA 2001 Workshop on Domain Specific Visual Languages*, Septembre 2001.

[10] Paul Clements and Linda Northrop. *Software product lines : practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[11] Krzysztof Czarnecki. *Generative Programming : Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technical University of Ilmenau, 1998.

[12] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software : An industrial experience report. In *GPCE '02 : Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 156–172, London, UK, 2002. Springer-Verlag.

[13] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Software Product Lines : Third International Conference, SPLC 2004*, pages 266–283. Springer-Verlag, 2004.

[14] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *In Proceedings of the 24th International Conference on Software Engineering (ICSE2002*, pages 266–276. ACM Press, 2002.

[15] Hassan Gomaa and Michael Eonsuk Shin. Multiple-view meta-modeling of software product lines. In *ICECCS '02 : Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, page 238, Washington, DC, USA, 2002. IEEE Computer Society.

[16] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* 1990.

[17] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. Form : A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5 :143–168, 1998.

[18] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*, pages 231–240, 2009.

[19] Daren Nestor, Luke O'Malley, Aaron Quigley, Ernst Sikora, and Steffen Thiel. Visualisation of variability in software product line engineering.

[20] Jean-Louis Peaucelle. Du dépeçage à l'assemblage, l'invention du travail à la chaîne de Chicago à Détroit. *Gérer et comprendre*, (73) :75–88, Septembre 2003.

[21] Schobbens Pierre-Yves, Heymans Patrick, Trigaux Jean-Christophe, and Bontemps Yves. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2) :456–479, 2007.

[22] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[23] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with uml multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, 2002.

[24] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Inf. Softw. Technol.*, 49(7) :717–739, 2007.

[25] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proc. of the 11th Intl. Conf. on Principles and Practiceof Constraint Programming (CP 2005)*, pages 827–831, Sitges, Spain, October 2005.

[26] Detlef Streitferdt, Matthias Riebisch, and Technische Universitat Ilmenau. Details of formalized relations in feature models using ocl. In *In Proceedings of 10th IEEE International Conference on Engineering of Computer Based Systems*, pages 297–304, 2003.

[27] Bednasch T. Konzept und implementierung eines konfigurierbaren metamodells fur die merkmalmodellierung, 2002. Available from `http://www.informatik.fh-kl.de/~eisenecker/studentwork/dt_bednasch.pdf` (en allemand).

[28] Jean-Christophe Trigaux and Patrick Heymans. Modelling variability requirements in software product lines : A comparative survey. Namur, Belgium, 2003.

[29] Jean-Christophe Trigaux and Patrick Heymans. Varied feature diagram (vfd) language : A reasoning tool. Technical report, PReCISE Research Center, University of Namur, Namur, Belgium, 2007.

[30] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *In Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE Computer Society, 2001.

[31] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000.

[32] Thomas von der Maßen and Horst Lichter. Requiline : A requirements engineering tool for software product lines. In *PFE*, pages 168–180, 2003.

[33] Wikipedia. General motors Wikipedia, 2010. `http://fr.wikipedia.org/wiki/General_Motors` [Online ; consulté le 09-Avril-2010].

Annexes

Annexe A : Code A-OCL permettant de calculer le prix total d'un ordinateur

Dans le code A-OCL ci-dessous, toutes les features de la Figure 3.6 ont été dotées d'un attribut réel "prix".

```
context Ordinateur, Carte_mère, Processeur, Carte_graphique, Accessoires
inv prix_Ordinateur :
Ordinateur:Prix = Carte_mère:Prix + Processeur:Prix + Carte_graphique:Prix
  + Accessoires:Prix

context Carte_mère, Asus, AOpen
inv prix_Carte_mère :
Carte_mère:Prix = Asus:Prix + AOpen:Prix

context Asus
inv prix_Asus :
if selected(Asus) then
Asus:Prix = 350
else
Asus:Prix = 0

context AOpen
inv prix_AOpen :
if selected(AOpen) then
AOpen:Prix = 250
else
AOpen:Prix = 0

context Processeur, CoreI7, Athlon
inv prix_Processeur :
Processeur:Prix = CoreI7:Prix + Athlon:Prix

context CoreI7
inv prix_CoreI7 :
if selected(CoreI7) then
CoreI7:Prix = 179.99
else
CoreI7:Prix = 0

context Athlon
inv prix_Athlon :
if selected(Athlon) then
Athlon:Prix = 99.99
else
Athlon:Prix = 0

context Carte_graphique, Nvidia, ATI
inv prix_Carte_graphique :
Carte_graphique:Prix = Nvidia:Prix + ATI:Prix
```

```
context Nvidia
inv prix_Nvidia :
if selected(Nvidia) then
Nvidia:Prix = 199
else
Nvidia:Prix = 0

context ATI
inv prix_ATI :
if selected(ATI) then
ATI:Prix = 99.99
else
ATI:Prix = 0

context Accessoires, Clavier_Souris, Ecran_Samsung, Ecran_Philips
inv prix_Accessoires :
if selected(Accessoires) then
Accessoires:Prix = Clavier_Souris:Prix + Ecran_Samsung:Prix + Ecran_Philips:Prix
else
Accessoires:Prix = 0

context Clavier_Souris
inv prix_Clavier_Souris :
if selected(Clavier_Souris) then
Clavier_Souris:Prix = 29.99
else
Clavier_Souris:Prix = 0

context Ecran_Samsung
inv prix_Ecran_Samsung :
if selected(Ecran_Samsung) then
Ecran_Samsung:Prix = 119.99
else
Ecran_Samsung:Prix = 0

context Ecran_Philips
inv prix_Ecran_Philips :
if selected(Ecran_Philips) then
Ecran_Philips:Prix = 99.99
else
Ecran_Philips:Prix = 0
```

Annexe B : Grammaire, règles de précédence et d'associativité de TVL

Grammaire de TVL

Voir page suivante.

```
----- Starting point ---------------------------------------------------------

MODEL = ( TYPE | CONSTANT | FEATURE )*

----- Type section -----------------------------------------------------------

TYPE =  SIMPLE_TYPE
      | RECORD

SIMPLE_TYPE =  "int" ID "in" SET_EXPRESSION ";"
             | "real" ID "in" SET_EXPRESSION ";"
             | "enum" ID "in" SET_EXPRESSION ";"
             | "int" ID ";"
             | "real" ID ";"
             | "bool" ID ";"

RECORD = "struct" ID "{" RECORD_FIELD+ "}"

RECORD_FIELD =  SIMPLE_TYPE
              | ID ID ";"

----- Constant section -------------------------------------------------------

CONSTANT =  "const" "int" ID INTEGER ";"
          | "const" "real" ID REAL ";"
          | "const" "bool" ID ( "true" | "false" ) ";"

----- ID section -------------------------------------------------------------
```

```
SHORT_ID =  "root"
          | "this"
          | "parent"
          | ID

LONG_ID =  SHORT_ID
         | SHORT_ID "." LONG_ID

----- Feature section ------------------------------------------------------

FEATURE =  "root" ID "{" FEATURE_BODY_ITEM+ "}"
         | LONG_ID "{" FEATURE_BODY_ITEM+ "}"
         | "root" ID FEATURE_GROUP
         | LONG_ID FEATURE_GROUP

FEATURE_BODY_ITEM =  DATA
                   | CONSTRAINT
                   | ATTRIBUTE
                   | FEATURE_GROUP

FEATURE_GROUP = "group" CARDINALITY "{" HIERARCHICAL_FEATURE ("," HIERARCHICAL_FEATURE)* "}"

HIERARCHICAL_FEATURE =  ("opt")? FEATURE
                      | ( "shared" | "opt" )? LONG_ID

CARDINALITY =  "oneof"
             | "someof"
```

```
              | "allof"
              | "[" NATURAL "," (NATURAL | "*" ) "]"

----- Attribute section ----------------------------------------------------------

ATTRIBUTE =  BASE_ATTRIBUTE
           | ID ID "{" SUB_ATTRIBUTE+ "}"

BASE_ATTRIBUTE =  "int" ID  ATTRIBUTE_BODY? ";"
                | "real" ID  ATTRIBUTE_BODY? ";"
                | "bool" ID  ATTRIBUTE_BODY? ";"
                | "enum" ID  ATTRIBUTE_BODY? ";"
                | ID ID ATTRIBUTE_BODY? ";"

ATTRIBUTE_BODY =  "is" EXPRESSION
                | "in" SET_EXPRESSION "," ATTRIBUTE_CONDITIONNAL
                | "in" SET_EXPRESSION
                | "," ATTRIBUTE_CONDITIONNAL

ATTRIBUTE_CONDITIONNAL  =  "ifin:" "is" EXPRESSION "," "ifout:" "is" EXPRESSION
                         | "ifin:" "is" EXPRESSION
                         | "ifout:" "is" EXPRESSION
                         | "ifin:" "in" SET_EXPRESSION "," "ifout:" "is" EXPRESSION
                         | "ifin:" "is" EXPRESSION "," "ifout:" "in" SET_EXPRESSION
                         | "ifin:" "in" SET_EXPRESSION "," "ifout:" "in" SET_EXPRESSION
                         | "ifin:" "in" SET_EXPRESSION
                         | "ifout:" "in" SET_EXPRESSION
```

```
SUB_ATTRIBUTE = ID ATTRIBUTE_BODY ";"

----- Expression section ----------------------------------------------------------

EXPRESSION =  EXPRESSION "&&" EXPRESSION
            | EXPRESSION "||" EXPRESSION
            | EXPRESSION "->" EXPRESSION
            | EXPRESSION "<-" EXPRESSION
            | EXPRESSION "<->" EXPRESSION
            | "!" EXPRESSION
            | "(" EXPRESSION ")"
            | "true"
            | "false"
            | LONG_ID
            | EXPRESSION "==" EXPRESSION
            | EXPRESSION "!=" EXPRESSION
            | EXPRESSION "<=" EXPRESSION
            | EXPRESSION "<" EXPRESSION
            | EXPRESSION ">=" EXPRESSION
            | EXPRESSION ">" EXPRESSION
            | EXPRESSION "in" SET_EXPRESSION
            | "and" "(" ( EXPRESSION_LIST | CHILDREN_ID ) ")"
            | "or" "(" ( EXPRESSION_LIST | CHILDREN_ID ) ")"
            | "xor" "(" ( EXPRESSION_LIST | CHILDREN_ID ) ")"
            | LONG_ID "excludes" LONG_ID
            | LONG_ID "requires" LONG_ID
            | EXPRESSION "+" EXPRESSION
            | EXPRESSION "-" EXPRESSION
```

```
              | EXPRESSION "/" EXPRESSION
              | EXPRESSION "*" EXPRESSION
              | "-" EXPRESSION
              | "abs" "(" EXPRESSION ")"
              | EXPRESSION "?" EXPRESSION ":" EXPRESSION
              | "sum" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
              | "mul" "(" ( EXPRESSION_LIST | CHILDREN_ID ) ")"
              | "min" "(" ( EXPRESSION_LIST | CHILDREN_ID ) ")"
              | "max" "(" ( EXPRESSION_LIST | CHILDREN_ID ) ")"
              | "count" "(" ( "children" | "selectedchildren" ) ")"
              | "avg" "(" ( EXPRESSION_LIST | CHILDREN_ID ) ")"
              | INTEGER
              | REAL

EXPRESSION_LIST = EXPRESSION ("," EXPRESSION_LIST)*

SET_EXPRESSION =  "{" EXPRESSION_LIST "}"
                | "[" SET_EXPRESSION_BOUND ".." SET_EXPRESSION_BOUND "]"

SET_EXPRESSION_BOUND =  INTEGER
                      | REAL
                      | "*"

CHILDREN_ID =  "selectedchildren" "." LONG_ID
             | "children" "." LONG_ID

----- Constraint section ---------------------------------------------------
```

```
CONSTRAINT =  "ifin:" EXPRESSION ";"
            | "ifout:" EXPRESSION ";"
            | EXPRESSION ";"

----- Data section ------------------------------------------------------------

DATA = "data" "{" DATA_PAIR+ "}"

DATA_PAIR = STRING STRING ";"

----- Value section -----------------------------------------------------------

NATURAL =  "0"
         | ["1"-"9"]["0"-"9"]*

INTEGER =  "0"
         | ("-")?["1"-"9"]["0"-"9"]*

REAL = INTEGER "." (["0"-"9"]*["1"-"9"])?

ID = ["a"-"z" "A"-"Z"] ["a"-"z" "A"-"Z" "0" -"9" "_"]+

STRING = " " " [^] " " "
```

Règles de précédence et d'associativité

Dans le tableau 1, les opérateurs des expressions sont cités par ordre de priorité croissant. Les opérateurs se situant au même niveau possèdent la même priorité. De plus, pour chaque opérateur, son associativité (gauche, droite ou non associatif) est aussi précisé

TABLE 1 – Règles de précédence et d'associativité

Associativité	Opérateur(s)
Droite	?
Droite	`<-`
Gauche	`->`
Non associatif	`<->`
Gauche	`||`
Gauche	`&&`
Non associatif	`==, !=, in`
Non associatif	`<=, <, >=, >`
Gauche	`+, -`
Gauche	`*, /`
Non associatif	`requires, excludes`
Droite	`!, -` (unaire), `abs(), min(), max(), count(), avg(), xor(), mul(), sum(), and(), or()`

Annexe C : Rapport de stage

Voir page suivante.

Rapport de stage

Support au développement de "Software Product Lines"

Faculté d'Informatique

FUNDP

Faber Paul

3 août 2010

Table des matières

1 Informations sur le lieu du stage

Research Center in Information Systems Engineering

University of Namur - Faculty of Computer Science Rue Grandgagnage 21,
B-5000 Namur, Belgium

Maître de stage :

Patrick Heymans
patrick.heymans@fundp.ac.be

Mon stage s'est déroulé dans les locaux de la faculté d'informatique au sein de l'équipe du LIEL. Cette équipe, dirigée par Mr. Heymans et Mr. Schobbens, s'intéresse notamment à divers aspects du domaine des Softwares Product Lines (SPL). De plus, ce groupe a publié de nombreux papiers en rapport avec ce même domaine. On peut donc dire qu'il est plus que compétent pour tout ce qui touche aux SPL. Finalement, il faut aussi préciser que cette équipe fait elle-même partie de PReCISE [1], un centre de recherche spécialisé dans l'ingénierie et la gestion des systèmes d'information avancés.

2 Objectifs fixés

Au tout début du stage, les objectifs n'étaient pas clairement fixés. Nous attendions surtout la venue d'un doctorant brésilien, Marcilio Mendonca [2], qui avait développé un outil [3] d'édition et de configuration de Feature Models (FMs). Comme son outil intéressait grandement Mr. Heymans et toute l'équipe, la venue du chercheur permettrait d'en savoir un peu plus, d'une part, sur son outil (architecture, possibilité d'intégration de son code,...) et d'autre part, sur le rôle que je pourrais jouer par rapport à ce dernier (développement d'un plug-in de visualisation, utilisation d'un nouveau type de solveur,...). Début septembre, la seule chose dont nous étions vraiment certain, c'est que le stage serait composé de deux grandes parties : une partie "analyse-recherche" et une

1. http ://www.fundp.ac.be/en/precise/
2. http ://csg.uwaterloo.ca/ marcilio/
3. http ://www.splot-research.org/

partie développement. Ainsi, Les objectifs "précis" ont été fixés au cours de l'évolution du stage. (Voir section 3)

3 Déroulement du stage et ajustement des objectifs

Avant de commencer le stage proprement dit, j'ai d'abord dû me documenter sur le domaine des Software Product Lines. A ce moment, il est vrai que je ne connaissais pas grand chose à ce sujet. Afin d'y remédier, j'ai dû lire une série d'articles [1] [2] [3] [6] [7] [8] préparée par Arnaud Hubaux.

Après cette lecture, et en attendant la venue de Marcilio Mendonca, je me suis concentré sur l'**analyse d'un outil de configuration de Feature Models** développé par deux jobistes au sein même de la faculté : "FeatureMX". Au final, cette étude devait permettre de mettre en avant les interactions possibles entre notre outil et celui développé par le chercheur Brésilien. L'analyse s'est déroulée en trois phases. La première phase fut surtout consacrée au code de l'outil : lecture du code des principales classes, lecture des diagrammes de classes, compréhension des patterns utilisés,... La seconde phase, quant à elle, a consisté en la rédaction d'un rapport d'architecture : modèle de données, descriptions des classes,... Enfin, la dernière phase concerna la rédaction d'un document des exigences du logiciel en respectant le template Volere [4], ce qui incluait par exemple : la description des différents acteurs, la définition d'un dictionnaire,... Cette première étape peut donc être considérée comme le **premier objectif** de mon stage.

Début octobre, Marcilio Mendonca est arrivé en Belgique. Lors de son séjour, nous avons fait quelques réunions dont une concernait l'architecture de son logiciel et du nôtre. Lors de cette dernière, j'ai brièvement présenté l'architecture de "FeatureMX" et le langage utilisé par celui-ci (VFD). Marcilio a fait de même pour son logiciel. Après discussion, nous sommes arrivés à la conclusion que l'architecture de "FeatureMX" était sans doute un peu trop complexe (sur-utilisation des patterns) et que son langage n'était pas assez simple et "userfriendly".

Après le départ de Marcilio, nous avons donc commencé à **définir un nouveau langage**. Pour ce faire, nous sommes partis d'un exemple de FM exprimé dans un langage concurrent et avons essayé de l'exprimer dans notre propre langage. A partir de là, nous avons donc pu voir les mécanismes manquants ou superflus de notre grammaire. Après plusieurs exercices de ce genre, nous sommes finalement arrivés à définir un tout nouveau langage appelé TVL [5], plus simple et plus "userfriendly". La définition de ce nouveau langage représente donc le **second objectif de mon stage**.

Dès lors, Andreas Classens et Quentin Boucher ont commencé à rédiger un papier [4] concernant TVL et sa sémantique (papier accepté au Workshop VaMoS [6]). De mon côté, j'ai été chargé du **développement d'un parseur (en Java) pour TVL**. Pour ce faire, j'ai utilisé deux bibliothèques : JFlex [7] et

4. http ://www.volere.co.uk/template.htm
5. http ://www.info.fundp.ac.be/ acs/tvl/
6. http ://www.vamos-workshop.net/2010/
7. http ://jflex.de/

Cup [8] (équivalents java de Lex et Yacc). La première grosse étape fut donc de définir une grammaire LALR non ambiguë. La seconde étape fut consacrée au développement du parseur lui-même. Ce dernier devait permettre de déjà faire du type checking : interdire la définition de plusieurs features root, s'assurer que les noms des features et des attributs respectent bien les conventions,... La conception de ce parseur peut donc être considérée comme le **troisième objectif du stage**.

Début décembre, lorsque le développement du parseur fut terminé, Patrick Heymans, Andreas Classens et moi-même avons fait une réunion pour fixer les **nouveaux objectifs** de mon stage. Lors de cette dernière, il a été décidé que je devrais ajouter trois nouvelles fonctionnalités au parseur : **la conversion d'un FM vers sa forme normalisée, la conversion de la forme normalisée vers la forme booléenne, et enfin, la soumission de cette forme booléenne à un solveur SAT** (en l'occurrence, SAT4J [9]). Chacun des objectifs devrait donc être atteint avant la fin de mon stage.

Suite à cette réunion, je me suis donc attaqué à la conversion d'un FM vers sa forme normalisée. Pour m'aider dans cette tâche, j'ai pu utiliser le rapport technique du langage TVL [5]. Tous les mécanismes de normalisation y étant expliqués en détail. Quelques jours avant Noël, j'ai eu terminé le développement de cette nouvelle fonctionnalité. J'ai profité du laps de temps restant avant les congés pour tester les versions normalisées des FMs produites par le logiciel.

A la rentrée, j'ai commencé la conversion de la forme normalisée vers la forme booléenne. Cette étape fut un peu plus délicate que la précédente vu que j'ai dû moi-même définir les formules booléennes équivalentes à certaines instructions du langage. Il y a donc eu deux grandes phases : une première durant laquelle j'ai défini les équivalences instructions du langage-formules booléennes, et une seconde durant laquelle j'ai implémenté toutes ces formules.

Enfin, quand j'ai eu terminé cette conversion, je me suis penché sur le dernier objectif : la soumission de la forme booléenne à un solveur SAT. Cette étape ne fut guère compliquée, il a suffi de transformer chaque formule booléenne en une formule DIMACS (voir section 4, sous-section *Utilisation d'un solveur SAT*) et d'ensuite fusionner ces dernières pour les soumettre au solveur.

En résumé (comme indiqué dans la section 2), le déroulement s'est composé de deux phases principales :

1. Une première phase "analyse-recherche", reprenant l'analyse de "FeatureMX" et la définition de TVL.
2. Une seconde phase de développement, reprenant l'implémentation du parseur et de ses fonctionnalités supplémentaires.

4 Activités scientifiques poursuivies durant le stage

Analyse du logiciel FeatureMX

La première activité scientifique de mon stage fut l'analyse du logiciel FeatureMX. A la base, ce programme devait permettre la configuration et l'édition

8. http ://www2.cs.tum.edu/projects/cup/
9. http ://www.sat4j.org/

de FMs. Durant mon étude, j'ai rédigé deux documents :

1. Un rapport d'architecture reprenant un modèle de données, une description des patterns utilisés, les diagrammes de classes, les descriptions des principales classes ainsi que les diagrammes de séquences.
2. Un rapport des exigences respectant le template Volere.

Durant cette analyse, j'ai donc pu mettre en application la théorie vue dans différents cours. Lors de la représentation des diagrammes, j'ai par exemple pu utiliser le cours d'analyse et modélisation des S.I de Mr. Heymans. Lors de la définition du data model, j'ai fait appel aux concepts vus lors du cours de modélisation pour l'informatique de Mr. Hainaut.

Définition d'un nouveau langage

Comme expliqué plus haut, après le séjour de Marcilio Mendonca, nous avons commencé à développer un nouveau langage de feature modelling. L'ancien langage (VFD), dont la syntaxe se rapprochait de celle du XML, ayant été jugé trop "lourd" et pas assez "userfriendly". Pour ce faire, je suis parti d'un FM défini dans un langage concurrent et ai tenté de le redéfinir en utilisant notre ancien langage. Cette première itération m'avait déjà permis de mettre plusieurs points en évidence : certains tokens n'étaient vraiment pas nécessaires, toute l'information ne pouvait pas être représentée de la manière la plus optimale,... En réitérant plusieurs fois ce processus, nous sommes finalement arrivés au résultat escompté : TVL, un nouveau langage de modélisation de la variabilité, plus simple et plus "userfriendly" que VFD.

Pour mettre en évidence une partie des progrès effectués depuis VFD, voici deux exemples. Le premier utilise la syntaxe de VFD :

```
:mand Vehicle
    <att> int price <val> :sel 10000
                          :desel 0
                  </val>
    </att>
    <gr> [1,1]
        :mand Ford,
        :mand Peugeot
    </gr>
```

Le second, celle de TVL :

```
Vehicle {
    int price, ifin: 10000, ifout: 0;
    group oneof {
        Ford,
        Peugeot
    }
}
```

Tous les tokens styles XML ont donc été abandonnés pour adopter une syntaxe similaire au C. Le résultat est donc plus léger et plus agréable à la lecture. Pour plus d'information à propos des avantages de TVL, se référer au papier [4].

Implémentation d'un parseur

Cette troisième activité scientifique marque la fin de la partie "analyse-recherche" du stage. Lors de cette activité, j'ai développé un parseur (en Java) pour TVL. Pour ce faire, j'ai utilisé deux bibliothèques, JFlex et Cup, les équivalents Java de Lex et Yacc. Les connaissances acquises lors du développement du compilateur en troisième baccalauréat m'ont été fort utiles. La tâche la plus ardue fut sans doute d'arriver à une grammaire LALR non ambigüe. Et ce, surtout que le langage continuait à évoluer. En plus du parseur même, j'ai aussi développé une table des symboles capable d'accueillir les données d'un arbre syntaxique (résultant du parsing d'un fichier TVL). Durant le développement de cette table, il a donc été nécessaire de faire du type checking. Par exemple : s'assurer qu'un attribut n'utilise pas des valeurs qui violeraient son type, vérifier qu'une feature ne possède pas plusieurs attributs avec un nom identique,...

Implémentation de la forme normalisée

L'implémentation de la forme normalisée représente une nouvelle fonctionnalité du parseur. Cette étape de développement ne fut pas la plus complexe. En effet, certaines transformations étaient déjà effectuées lors de la création de la table des symboles. De plus, le rapport technique du langage TVL contenait la liste de toutes les transformations nécessaires pour arriver à un FM normalisé [5]. Par exemple, l'attribut :

```
size carSize;
```

utilisant le type :

```
struct size {
    int height;
    int width
}
```

est normalisé de cette façon :

```
int carSize_Height;
int carSize_Width;
```

Globalement, la normalisation d'un FM (voir figure 1) se déroule comme suit :

1. A partir de la table des symboles du fichier base, un nouvel arbre syntaxique est créé. C'est durant cette étape que toutes les transformations nécessaires à la normalisation sont effectuées. L'arbre créé contient la version normalisée du FM (partir de la table des symboles permet de s'assurer que le modèle de base ne contient pas d'erreurs).
2. L'arbre syntaxique nouvellement créé est ensuite transformé en un nouveau fichier TVL.
3. Tout comme pour un fichier normal, le fichier contenant la forme normalisée est parsé. Cette étape permet de s'assurer que la version normalisée du FM ne contient pas d'erreurs.

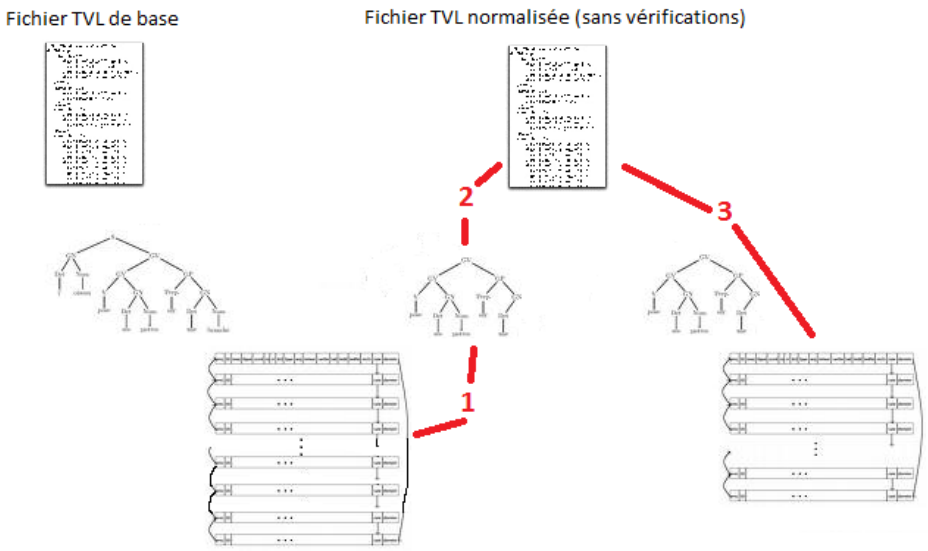

FIGURE 1 – Processus de normalisation d'un FM.

Implémentation de la forme booléenne

La transformation de la forme normalisée d'un FM vers sa forme booléenne se révéla un peu plus délicate que l'activité précédente. En effet, elle nécessita plus de travail dans le code. De plus, la plupart des transformations permettant de passer d'une instruction du langage vers la formule booléenne équivalente n'étaient pas définies. J'ai donc d'abord dû définir ces transformations pour ensuite seulement pouvoir commencer l'implémentation même. Par exemple, l'instruction correspondant à l'attribut :

```
enum color in {Red, Blue};
```

est transformée comme suit :

```
bool color_Red;
bool color_Blue;
(color_Red && !color_Blue) || (color_Blue && !color_Red);
```

Le processus de mise sous forme booléenne est quasi identique à celui de la normalisation (voir figure 1) sauf que la forme booléenne ne contiendra bien sûr que des attributs booléens.

Utilisation d'un solveur SAT

La dernière activité consista à transformer la forme booléenne pour ensuite la soumettre à un solveur SAT. Pour ce faire, j'ai utilisé SAT4J. Ce type de solveur n'accepte que des formules sous format DIMACS, c'est-à-dire des formules en CNF ne faisant intervenir que des variables identifiées par un entier. Vu que lors de la transformation en forme booléenne, toutes les contraintes sont mises en CNF, il m'a suffit de remplacer chaque nom de feature ou d'attribut par

un identifiant numérique unique. De plus, il m'a également fallu produire les formules DIMACS correspondants aux cardinalités et aux règles de justification de ces dernières.

Actuellement, le solveur est capable de calculer différentes informations par rapport à un FM :

- La satisfiabilité.
- Le nombre de produits valables et la liste de ces derniers.
- Indiquer si une feature est morte (i.e. présente dans aucun produit).
- ...

5 Auto-évaluation

Un point fort positif fut mon intégration au sein de l'équipe du LIEL. Au début de mon stage, je ne connaissais pas vraiment les membres du bureau. Au fur et à mesure, j'ai dû apprendre à m'intégrer et à connaître chacun. L'étape clé fut sans aucun doute la définition du nouveau langage, cette dernière m'a permis de me rendre compte à quel point il était nécessaire de savoir travailler en groupe : partager son avis, accepter la critique, savoir écouter les autres... Cette facette du stage m'a vraiment parue motivante et réussie.

La seconde note positive de mon stage concerne le langage Java et l'environnement de développement Eclipse. Tout au long de mon cursus, j'ai du utiliser ces deux produits dans différents contextes (MDL, projet de troisième baccalauréat,...). Durant mon stage, implémenter un programme en Java tout en utilisant Eclipse ne posa donc aucun problème et me permit même d'acquérir de l'expérience supplémentaire.

Ensuite, si je devais améliorer quelque chose, ce serait sans doute ma maîtrise de l'anglais. Quand Marcilio Mendonca est arrivé en Belgique, j'ai pu mesurer la différence entre mon anglais et celui pratiqué au sein de l'équipe de travail. Je trouvais ça vraiment dommage de ne pas pouvoir tout comprendre lors des discussions. Dès lors, je n'osais pas toujours intervenir car je craignais d'avoir mal compris quelque chose. Bien que ces échanges n'aient vraiment duré qu'une semaine, si je devais recommencer mon stage, j'essaierais de mieux m'y apprêter.

Parallèlement, je corrigerais aussi ma façon de coder. En effet, quand je suis lancé dans l'implémentation d'une méthode, je ne m'arrête pas toujours pour faire le point sur ce que je viens de coder. Dès lors, quand je lance la compilation, il est parfois frustrant de voir les erreurs de distraction que j'ai pu laisser derrière moi. De plus, je ne prends pas toujours le temps de directement documenter les classes et méthodes que je viens de créer. Quelques fois, je suis donc obligé de revenir dans une classe et de me replonger dans son code pour pouvoir la documenter. Donc, dorénavant, j'essaierai de ne pas vouloir aller trop vite et de directement documenter mon code.

Maintenant que j'ai pu avoir une vue globale de mon stage et des tâches que j'y ai effectuées, je pense que l'on peut dire que tous les objectifs ont été remplis [10]. Ceci, je le dois à toute l'équipe qui m'entourait, chacun étant toujours disponible et ne rechignant jamais à répondre à l'une des mes questions. De plus, je dois dire que l'ambiance de travail était très professionnelle tout en étant décontractée.

10. Cependant, un informaticien n'étant jamais à l'abris d'un bug, je resterai toujours à disposition de Mr. Heymans en cas d'erreur dans le programme que j'ai développé.

Grâce à ce stage, j'ai aussi pu avoir une premier aperçu de la vie professionnelle et de tout ce qu'elle incombe : le respect des horaires, les échanges entre les membres d'une équipe, le respect des deadlines,... Cette expérience unique m'a sans doute permis de mieux me préparer au futur travail d'informaticien qui m'attend.

Pour terminer, je tiens à remercier Mr. Heymans et toute l'équipe du LIEL pour l'accueil et le support qui m'ont été donnés.

Références

[1] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models : challenges ahead. *Commun. ACM*, 49(12) :45–47, 2006.

[2] Don S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.

[3] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feature models. In *JISBD*, pages 367–376, 2006.

[4] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 159–162. University of Duisburg-Essen, January 2010. Acceptance rate : 54

[5] Andreas Classen, Quentin Boucher, Paul Faber, and Patrick Heymans. Syntax and semantics of TVL, a text-based feature modelling language. Technical report, PReCISE Research Center, University of Namur, Namur, Belgium, 2010. www.info.fundp.ac.be/∼acs/tvl.

[6] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. Formal modelling of feature configuration workflows. In John D. McGregor and Dirk Muthig, editors, *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*, pages 221–230. SEI, Carnegie Mellon University, 2009. Acceptance rate : 37

[7] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines : A separation of concerns, formalization and automated analysis. In *RE*, pages 243–253, 2007.

[8] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2) :456–479, 2007.

Annexe D : Règles vérifiées durant l'analyse sémantique

Règles concernants les identifiants

La première de ces règles concerne la forme des identifiants. Ceux correspondants à des features sont obligés de commencer par une majuscule tandis que tous les autres (ceux concernants des attributs, types et constantes) doivent débuter par une minuscule.

Ensuite, il existe aussi des contraintes par rapport au caractère unique de certains identifiants :

- L'identifiant de la feature racine doit être unique, aucune autre feature ne peut être nommée de la même façon.
- Les attributs d'une même feature doivent porter des noms uniques.
- Les features enfants d'une même feature parente doivent porter des noms uniques. Deux features soeurs (faisant partie du même groupe de features enfants) ne peuvent donc être identifiées de la même façon.
- Les identifiants des types et des constantes doivent être uniques. Un attribut ne pourra par exemple pas posséder le même identifiant que celui d'un type.
- Les types simples d'un même type structuré doivent porter des noms uniques.
- Les valeurs d'un même attribut énumératif doivent être uniques.
- Une valeur d'un attribut énumératif ne peut être identique à un identifiant (de feature, de constante, ...).

Un identifiant ou un chemin d'identifiants doit toujours être utilisé de manière non-ambiguë, il ne peut référencer qu'un et un seul élément. En outre, il doit aussi bien sur faire référence à un élément existant.

Enfin, il existe un ensemble de mots réservés ("`group`", "`int`", "`bool`", ...) qui ne peuvent être utilisés pour identifier un élément.

Règles concernants la structure d'un modèle

Les features enfants ne peuvent être spécifiées que dans une et une seule déclaration de leur feature parente. En outre, avant de pouvoir être déclarée individuellement (hors d'un groupe de features enfants), une feature doit avoir été déclarée dans un groupe de features enfants (cette règle n'est évidemment pas d'application pour la feature racine).

Le domaine de valeurs d'un attribut ne peut être défini qu'une et une seule fois, soit dans la déclaration de l'attribut, soit dans la déclaration du type de l'attribut (si ce dernier est défini à l'aide d'un type créé par l'utilisateur). Dans le cas où la valeur de l'attribut est précisée (mot-clé "`is`"), il ne peut y avoir de blocs conditionnels ("`ifin:`" ou "`ifout:`") car cette valeur est considérée comme constante (que la feature de l'attribut soit sélectionnée ou pas, l'attribut gardera toujours cette valeur).

De par sa nature de graphe orienté acyclique, un modèle ne peut pas contenir de cycles entre ses features. Une feature parente ne peut donc jamais devenir (directement ou indirectement) la fille d'une de ses features enfants. De plus, dans la déclaration d'une feature partagée, le mot-clé "`parent`" ne peut jamais

être utilisé. Si c'était le cas, il pourrait faire référence à plusieurs features et il serait donc ambigu.

La cardinalité d'un groupe de features enfants doit toujours être logique, c'est-à-dire :

- Que les valeurs des bornes doivent être entières et non négatives.
- Que la valeur de la borne minimum doit être inférieure ou égal à la borne maximum.
- Que la valeur de la borne maximum doit être inférieure ou égal au nombre total de features du groupe.

Enfin, il est interdit de déclarer un type structuré à l'intérieur d'un autre type structuré.

Règles concernants les types

Les expressions et les ensembles rentrant dans la définition d'un attribut doivent être compatibles avec le type de ce dernier. Par exemple, un attribut de type réel ne pourra être défini qu'avec des expressions et/ou des ensembles de type réel ou entier.

Toutes les contraintes doivent être de type booléen.

Chaque expression ou ensemble utilisé dans la définition d'un attribut ou d'une contrainte doit être lui-même être correctement typé. Les opérateurs numériques ne peuvent donc être employés qu'avec des expressions numériques, les opérateurs de comparaison ne peuvent être utilisés qu'avec deux expressions dont les types sont compatibles entre eux, ... Quant aux ensembles, ils ne peuvent contenir que des valeurs compatibles entre elles (un ensemble ne pourra par exemple pas contenir à la fois des strings et des entiers) et avec le type de l'attribut pour lequel ils sont utilisés. De plus, en ce qui concerne les intervalles, la borne minimum doit bien sur être inférieure ou égal à la borne maximum.

L'utilisation des mots-clés "`selectedchildren`" et "`children`" ne peut se faire qu'avec un attribut commun à toutes les features enfants d'une feature parente. Le type de cette attribut devra être identique pour chaque feature enfant et compatible avec la fonction d'agrégation utilisée.

Annexe E : Liste des transformations de la normalisation

Les règles [8] reprises ci-dessous permettent de transformer un modèle TVL en son équivalent en forme normale. Chacune de ces règles est numérotée car il est important de les appliquer en suivant l'ordre de cette numérotation. Dans le cas contraire, le modèle en forme normale pourrait contenir des erreurs.

1) Il faut éliminer toutes les instructions "`include`" en les remplaçant par le contenu des fichiers qu'elles référencent.

2) Il faut éliminer toutes les constantes "`const t c e`" en remplaçant chaque occurence de "`c`" par "`e`".

3) Il faut éliminer tous les types utilisés. Dans les cas d'un type simple "`b t`", il faut remplacer le type de chaque attribut utilisant "`t`" par le type de base "`b`". Dans le cas d'un type structuré "`struct t {b1 t1, b2 t2, ... }`", il faut procéder en plusieurs étapes. Durant la première, le type structuré "`t`" est décomposé et remplacé par l'ensemble des types simples le constituant "`b1 t_t1; b2 t_t2; ...`". Ensuite, les attributs utilisants le type "`t`" sont eux aussi décomposés et remplacés par des attributs individuels utilisants les types simples "`t_t1, t_t2, ...`" Par après, ces types simples sont eux aussi éliminés à leur tour.

4) Il faut éliminer les domaines de valeurs et les valeurs des attributs. Dans le cas d'un attribut "`t a in s`", la clause "`in`" est supprimée et la nouvelle contrainte "`this.a in s`" est ajoutée. Dans le cas d'un attribut "`t a is v`", la clause "`is`" est supprimée et la nouvelle contrainte "`this.a == v`" est ajoutée.

5) Il faut éliminer les blocs conditionnels des attributs. Si l'attribut "`t`" possède un bloc "`ifin: is v`" ou "`ifin: in s`", le bloc est supprimé et la contrainte "`ifin: t == v`" ou la contrainte "`ifin: t in s`" est ajoutée. De façon similaire, si "`t`" possède un bloc "`ifout: is v`" ou "`ifout: in s`", le bloc est supprimé et la contrainte "`ifout: t == v`" ou la contrainte "`ifout: t in s`" est ajoutée.

6) Il faut éliminer les contraintes conditionnelles. Chaque contrainte "`ifin: e`" ou "`ifout: e`" est remplacée par une contrainte équivalente "`this -> e`" ou "`!this -> e`".

7) Il faut éliminer les mots-clés "`children`" et "`selectedchildren`" en (ici, les $c_1, ..., c_k$ sont les features enfants de la feature où est utilisée la fonction d'agrégation) :

- Remplaçant "`avg(children.a)`" par "`sum(children.a) / count(children)`". Pour "`selectedchildren`", la transformation est similaire.
- Remplaçant "`fct(children.a)`" par "`fct(`c_1`.a, ..., `c_k`.a)`" où "`fct`" peut être "`sum`", "`mul`", "`min`", "`max`", "`and`", "`or`" ou "`xor`".
- Remplaçant "`count(children)`" par le nombre de features enfants de la feature où "`count`" est utilisé.

- Remplaçant "`count(selectedchildren)`" par "`sum((`c_1 `? 1 : 0), ..., (`c_k `? 1 : 0))`"
- Remplaçant "`fct(selectedchildren.a)`" par "`fct((`c_1 `? `c_1`.a : neut), ..., (`c_k `? `c_k`.a : neut))`" où "`fct`" peut être "`sum`", "`mul`", "`min`", "`max`", "`and`", "`or`" ou "`xor`" et "`neut`" est l'élement neutre de la fonction d'agrégation (0 pour l'addition, 1 pour la multiplication, `true` pour la conjonction, `false` pour la disjonction et pour "`xor`").

8) Il faut éliminer les mots-clés "`this`", "`parent`" et "`root`" en les remplaçant par les identifants ou les chemins d'identifiants non ambigus leurs correspondants.

9) Il faut fusionner toutes les contraintes du modèle en une seule et même contrainte et rattacher cette dernière à la feature racine r. Cette contrainte globale correspond à Φ.

10) Il faut éliminer les mots-clés des cardinalités. Ainsi, chaque occurence de :
- "`oneof`" doit être remplacée par "`group [1..1]`".
- "`someof`" doit être remplacée par "`group [1..*]`".
- "`allof`" doit être remplacée par "`group [*..*]`".

11) Il faut regrouper les différentes déclarations d'une même feature dans une seule déclaration. Cette déclaration globale doit être insérée dans le groupe de features enfants où la feature est déclarée.